2019

Attacking and securing network time protocol

BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

ATTACKING AND SECURING NETWORK TIME PROTOCOL

by

AANCHAL MALHOTRA

M.S., Boston University, 2015

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2019

Approved by

First Reader

Sharon Goldberg, PhD
Associate Professor of Computer Science

Second Reader

Ran Canetti, PhD
Professor of Computer Science

Third Reader

Leonid Reyzin, PhD
Professor of Computer Science

Acknowledgements

First and foremost I want to thank my advisor Prof. Sharon Goldberg for being a tremendous mentor. Since the beginning of my graduate school, Sharon believed in me like nobody else and gave me unwavering support throughout. The joy and enthusiasm she brings to research is contagious and has been a great motivation, particularly during tough times in my Ph.D. pursuit. I am also grateful for the excellent example she has set as a strong woman, a successful professor and a wonderful mother.

I'd like to give a big shout out to Prof. Leonid Reyzin and Mayank Varia for being my "second" mentors. I am grateful to Leo for providing indespensable advise in almost every aspect of my research and personal life. I thank Mayank who taught me how to think like an applied cyptographer and for introducing me to the technical tools needed to do my research. Mayank's ability to offer an extremely clear exposition of almost any topic simply amazes me. I especially thank Prof. Steve Homer for always looking out for me and helping me navigate the processes of Ph.D. program efficiently.

I would not have achieved this milestone without my extremely competent and awesome co-authors. I have thoroughly enjoyed the process of coming up with hard questions and brainstorming solutions together, which incented me to widen my research from various perspectives.

I am also grateful to my industry collaborators and IETF pals. I spent a fantastic summer as a security research intern at NLnet Labs, Amsterdam where my mentors Willem Toorop and Benno Overeinder went above and beyond to teach me the ins and outs of DNS protocol. I was also fortunate to have spent three months as cryptography engineer at Cloudflare, San Francisco. I am extremely thankful to my mentor Nick Sullivan there for making the seemingly hard task of taking an idea from inception to the running code in production, a smooth and enjoyable experience. I am also

grateful to Nick for being a part of my Ph.D. committee and providing insightful comments on my thesis. I have benefited substantially also from my conversations with several experts at IETF including Allison Mankin, Daniel Franke, Harlan Stenn, Jared Mauch, Karen O'Donoghue, Miroslav Lichvar, Rich Salz, Suresh Krishnan, Tal Mizrahi and others on various aspects of Internet protocols and security and the complexities involved in standardization processes.

My time at Boston University would not have been as much fun and worthy without my friends Ethan Heilman, Nabeel Akhtar, Oxana Poburinnaya, Sarah Scheffler, Tommy Unger, and William Koch, Yilei Chen.

I cannot but express my deep gratitude to my partner Abhishek Sharma for always being my critic and for holding up with me during my lows. You are my biggest inspiration and the sole reason that I started my journey as a researcher in the first place.

Last but not the least I'd like to thank my parents Poonam and Harish and my brother Parul for believing in me and supporting me throughout this long and eventful journey.

I have thoroughly enjoyed my Ph.D. student life mainly because of all the amazing people around me.

ATTACKING AND SECURING NETWORK TIME PROTOCOL

AANCHAL MALHOTRA

Boston University, Graduate School of Arts and Sciences, 2019

Major Professor: Sharon Goldberg, Associate Professor of Computer Science

ABSTRACT

Network Time Protocol (NTP) is used to synchronize time between computer systems communicating over unreliable, variable-latency, and untrusted network paths. Time is critical for many applications; in particular it is heavily utilized by cryptographic protocols. Despite its importance, the community still lacks visibility into the *robustness* of the NTP ecosystem itself, the *integrity* of the timing information transmitted by NTP, and the *impact* that any error in NTP might have upon the security of other protocols that rely on timing information. In this thesis, we seek to accomplish the following broad goals:

1. Demonstrate that the current design presents a security risk, by showing that network attackers can exploit NTP and then use it to attack other core Internet protocols that rely on time.

2. Improve NTP to make it more robust, and rigorously analyze the security of the improved protocol.

3. Establish formal and precise security requirements that should be satisfied by a network time-synchronization protocol, and prove that these are sufficient for the security of other protocols that rely on time.

We take the following approach to achieve our goals incrementally.

1. We begin by (a) scrutinizing NTP's core protocol (RFC 5905) and (b) statically analyzing code of its reference implementation to identify vulnerabilities in protocol design, ambiguities in specifications, and flaws in reference implementations. We then leverage these observations to show several off- and on-path *denial-of-service* and *time-shifting attacks* on NTP clients. We then show *cache-flushing* and *cache-sticking* attacks on DNSSEC that leverage NTP. We quantify the attack surface using Internet measurements, and suggest simple countermeasures that can improve the security of NTP and DNS(SEC).

2. Next we move beyond identifying attacks and leverage ideas from Universal Composability (UC) security framework to develop a *cryptographic model* for attacks on NTP's datagram protocol. We use this model to prove the security of a *new backwards-compatible* protocol that correctly synchronizes time in the face of both off- and on-path network attackers.

3. Next, we propose general security notions for network time-synchronization protocols within the UC framework and formulate ideal functionalities that capture a number of prevalent forms of time measurement within existing systems. We show how they can be realized by real-world protocols (including but not limited to NTP), and how they can be used to assert security of time-reliant applications — specifically, cryptographic certificates with revocation and expiration times. Our security framework allows for a clear and modular treatment of the use of time in security-sensitive systems.

Our work makes the core NTP protocol and its implementations more robust and secure, thus improving the security of applications and protocols that rely on time.

Contents

List of Tables

List of Figures

List of Abbreviations

CVE	.....	Common Vulnerabilities and Exposures
DoS	.....	Denial-of-Service
DNS	.....	Domain Name System
DNSSEC	.....	DNS Security Extensions
HSTS	.....	HTTP Strict Transport Security
HTTP	.....	Hyper Text Transfer Protocol
IETF	.....	Internet Engineering Task Force
IP	.....	Internet Protocol
KoD	.....	Kiss-o'-Death
MAC	.....	Message Authentication Code
MiTM	.....	Man-in-The-Middle
NTP	.....	Network Time Protocol
ntpd	.....	Network Time Protocol daemon
PKI	.....	Public Key Infrastructure
PTP	.....	Precision Time Protocol
RFC	.....	Request For Comments
RPKI	.....	Routing Public Key Infrastructure
SNTP	.....	Simple Network Time Protocol
TLS	.....	Transport Layer Security
TTL	.....	Time To Live
UC	.....	Universal Composability
UDP	.....	User Datagram Protocol

Chapter 1

Introduction

1.1 Time Synchronization & Network Time Protocol (NTP)

The need for time in computer network systems does not arise from the desire to provide synchronous communication, quality of service, or other "sophisticated" networking primitives. Rather, the need for correct time is often coupled with the safe use of applications especially for cryptography to thwart attacks against the network.

It may appear that *measuring* real time is a relatively easy task; indeed, most computing platforms today, even low end ones, are equipped with a built-in clock. Still, synchronizing and adjusting these clocks, and in particular reaching agreement on time in an asynchronous network like the Internet turns out to be non-trivial. In particular, Network Time Protocol (NTP), one of the oldest protocols in the Internet and the current Internet Engineering Task Force (IETF) standard to *synchronize time* between computer systems communicating over unreliable variable-latency network paths, is rather complex. In its most typical client-server mode of operation, NTP assumes a hierarchical system of "time servers", where lower-stratum servers are assumed to have a more accurate notion of time, and higher-stratum servers determine time by querying several lower-stratum ones and performing some complex aggregation of the responses. The protocol has mechanisms for protecting from errors introduced by network delays, but is built on complete trust in the queried time servers, as well as in the authenticity of the communication.

The goal of this thesis is to make the core NTP protocol and its implementations

more robust and secure.

1.2 Motivation to Secure NTP

Time matters! Time is a fundamental building block for multiple applications; in particular it is heavily utilized by cryptographic protocols. For instance, cryptographic protocols use timestamps to prevent replay attacks and limit the use of stale or compromised cryptographic material, *e.g.,* TLS (Rescorla, 2018), HSTS (Hodges et al., 2012), DNSSEC (Arends et al., 2005a), RPKI (Lepinski and Kent, 2012), authentication protocols, etc. while accurate time synchronization is a basic requirement for various distributed systems like email, bitcoin (Nakamoto, 2008), etc.

Why do we care about NTP? NTP is the most dominant protocol in the Internet to update system time (Minar, 1999), (Murta et al., 2006), (Czyz et al., 2014a) (Mauch, 2015). When NTP fails on the system, multiple applications on the system can fail, all at the same time. Such *benign* (non-malicious) failures have happened. On November 19, 2012 (Bicknell, 2012), for example, two important NTP (stratum 1) servers, `tick.usno.navy.mil` and `tock.usno.navy.mil`, went back in time by about 12 years, causing outages at a variety of devices including Active Directory (AD) authentication servers, PBXs and routers (Morowczynski, 2012). Other than these benign failures, exploits of individual NTP clients can also serve as a building block for *malicious* attacks on other protocols and applications. For example, several authors (Mills, 2011, pg 33, pg 183) (Klein, 2013), (Selvi, 2015) have observed that NTP could be used to undermine the security of TLS certificates. Others like (Corbixgwelt, 2011) suggest how an NTP attacker can trick a victim into rejecting a legitimate block on bitcoin blockchain, or waste computational power on a stale block. (Klein, 2013) also discusses the implications of shifting time on logging and authentication services. (Selvi, 2014), (Selvi, 2015) was the *first*

Table 1.1: Attacking various applications with NTP.

To attack...	change time by ...	To attack...	change time by ...
TLS Certificates	years or months	Routing (RPKI)	days
HSTS	a year	Bitcoin	hours
DNSSEC	months	API authentication	minutes
Kerberos	minutes	DNS Caches	days

to practically demonstrate Man-in-The-middle timeshifting attacks on Simple NTP (SNTP) (Mills, 2006) to attack another application HTTP Strict Transport Security (HSTS). HSTS is used by a webserver to tell its client that all future connections should use HTTPS. The HSTS request, however, comes with an expiry time (one year is recommended (OWASP, 2015)) after which point the client need no longer use an encrypted connection; thus, an NTP attacker that sends the client a year into the future can effectively disable HSTS, allowing for downgrade attacks (*e.g.,* 'SSL Stripping' (Marlinspike, 2009)) to non-encrypted connections.

We summarize (in Table 1.1) various applications that can be subverted by leveraging vulnerabilities in NTP. This further motivates the need for securing NTP.

1.3 Our Novel Research Approach Towards Network Protocol Security

In this thesis we use a novel research approach towards network protocol security comprising of techniques ranging from *theoretical* to *applied*. While we focus on securing NTP, our research approach is general and can be applied to other network protocols.

Overview of network protocols ecosystem (See Figure 1·1) Network protocols like NTP are specified in the documents called RFCs published by IETF, the standards body for Internet protocols. These are important documents that are referenced by different developers for various implementations. A single protocol may

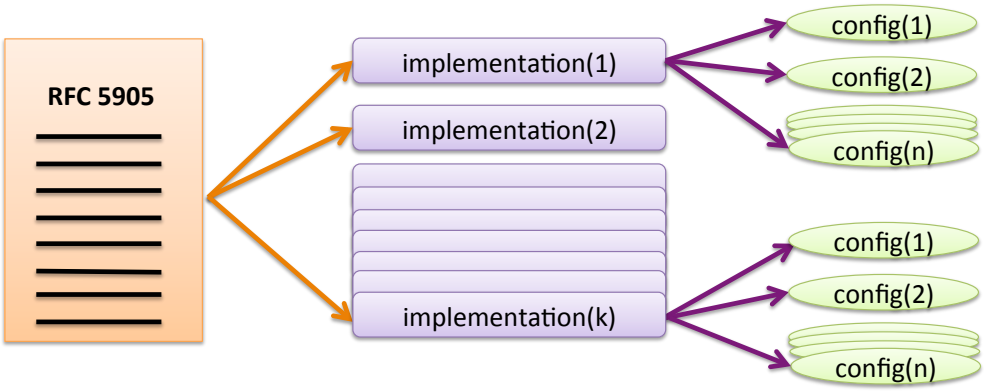

Figure 1·1: Overview of network protocols ecosystem.

have several implementations. Further each implementation may have several different configurations[1] when they are deployed in the real world. Although these implementations when shipped with operating systems have certain default configuration, they may be customized by the end users.

Following is the exposition of our approach towards network protocols security:

Scrutinizing RFCs. In order to do any meaningful research on network protocols, the first obvious step is to take a deep dive into RFCs. This helps us recognize *underspecifications* in the protocol and *ambiguities* in the written language that may lead to vulnerabilities.

Analyzing implementation code. The next step is to analyze the code of different *implementations* of the protocol to see how these underspecifications and ambiguities are interpreted by the developers and baked into the software[2].

This two-step process exposes potential security vulnerabilities due to flaws in protocol design or gaps in implementation and specifications.

Simulating attacks and network measurements. The next step is to demonstrate network attacks on the protocol for different implementations that leverage

[1]Implementations come with configuration files used to configure the parameters and initial settings for some computer programs.

[2]RFCs do not necessarily mandate everything. Some things are left to the interpretation or choice of developers.

vulnerabilities identified in the above process. Finally we perform measurements in the Internet to determine the attack surface *i.e.,* the number of IPs that are vulnerable to our attacks.

Security & Cryptography. Having demonstrate the fragility/subvertibilty of the protocol, the next step is to make it more robust and secure. Our approach towards this depends on the underlying issue. Sometimes these vulnerabilities are due to gaps between the specifications and implementations. This usually requires modifications in RFCs and code changes in the existing software. Other times the identified security issue is due to flaw(s) in the design of the core protocol itself, which requires a change in the protocol design and use of cryptographic tools to prove its security.

1.4 NTP Security: History & Current-State-of-Art

NTP's design rationale. At the time of its inception back in 1985 (Mills, 1985, Sec. 2) there were two main design goals for the service provided by NTP.

1. *Robustness* To mitigate errors due to equipment or network propagation failures, NTP was designed as a service where a client can gather timing samples from multiple peers over multiple communication paths and then combine them to get more accurate measurement.

2. *Load distribution* Top-level NTP servers that are directly attached to high precision time-keeping devices like atomic clocks, GPS, etc have more accurate time. While ideally every NTP client would like to connect to these devices, that would wreck havoc for these relatively fewer NTP servers and the network. So to reduce protocol load on the network, NTP service was designed in a hierarchical manner.

RFC 958 (Mills, 1985, Sec. 3) also states the following non-goals of the protocol:

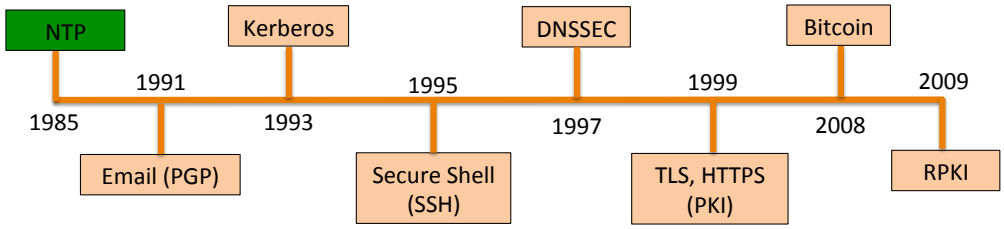

Figure 1·2: Timeline of development of protocols after NTP.

1. *Peer authentication* The protocol does not provide any guarantee on the identity of the entity that the NTP client thinks that it is connected to.

2. *Data integrity* The protocol provides no guarantee on the correctness of the data delivered. Integrity check was deferred to the lower layers of the protocol stack such as IP and UDP layer checksums.

It is clear from above that security against malicious entities was a "non-goal" for the protocol at the time of its design. We attribute this to the following two reasons:

1. Like many other protocols in common use on the Internet today, NTP was designed at a time when Internet was not as vast and full of untrusted entities as it is today. *e.g.,* until late 1980's, before the commercialization of the Internet with the arrival of ISPs and World Wide Web (WWW), the Internet was basically an internetwork of few mutually-trusted nodes (Leiner et al., 1999).

2. Time, unlike today, was not crucially used by other security critical applications. See, for example, Figure 1·2 for a rough timeline of the development of security protocols, that depend on time to provide functionality and security guarantees, that developed after NTP.

Current state-of-art of NTP security. Since the need for securing NTP was realized after the development of protocols that relied on time, there were proposals for cryptographically-authenticating NTP[3]. While NTP supports both symmetric (Mills et al., 2010) as well as asymmetric authentication (Haberman and Mills, 2010) mechanisms, we found that they are rarely used in practice because of limitations of both.

Symmetric cryptographic authentication appends an MD5 hash keyed with symmetric key k of the NTP packet contents m as MD5($k||m$) (Mills, 2011, pg 264) to the NTP packet. MD5($k||m$) is intended to be a cryptographic message authentication code (MAC). There are at least two major problems with this approach. *First*, the use of MD5 in this manner is not a provably secure MAC and is also vulnerable to length-extension attacks (Bellare et al., 1996); HMAC should be used instead (Bellare et al., 1996). Moreover, MD5 is not collision-resistant (Wang and Yu, 2005) and has been deprecated in favor of more secure hash functions like SHA-256 (Turner and Chen, 2011). *Second*, the symmetric key k must be pre-configured manually, which makes this solution quite cumbersome for public servers that must accept queries from arbitrary clients. (Indeed, NIST operates important public stratum 1 servers and distributes symmetric keys only to users that register, once per year, via US mail or facsimile (NIST, 2010); the US Naval Office does something similar (USNO, 2015).)

Asymmetric cryptographic authentication is provided by the Autokey protocol, first described in RFC 5906 (Haberman and Mills, 2010). RFC 5906 is not a standards-track document (it is classified as 'Informational'), NTP clients do not request Autokey associations by default (Autokey, 2012), and many public NTP servers do not support Autokey (*e.g.,* the NIST timeservers (NIST, 2010), many servers in

[3]The payload of NTP packets consists basically of timestamps, which are not considered secret (Mizrahi, 2012b). Therefore, encryption of the time synchronization protocol packet's payload is usually of low importance.

`pool.ntp.org`). The most critical problem, however, is that the protocol has non-standard authentication mechanisms and is badly broken. Any network attacker can trivially retrieve the secret key shared between the client and the server (Röttger, 2012). In fact, a lead developer of the ntpd client wrote in 2015 (Stenn, 2015a): "Nobody should be using autokey. Or from the other direction, if you are using autokey you should stop using it."

These limitations inhibit the adoption of cryptographically-authenticated NTP, thus exposing it to network attacks.

1.5 Our Goals

We observe that despite the critical role played by NTP, the community lacks visibility into the *robustness* of the NTP ecosystem itself, the *integrity* of the timing information transmitted by NTP, and the *impact* that any error in NTP might have upon the security of other protocols that rely on timing information. In this thesis, we seek to accomplish the following broad goals:

1. Demonstrate that the current design presents a security risk, by showing that network attackers can exploit NTP and then use it to attack other core Internet protocols that rely on time.

2. Improve NTP to make it more robust, and rigorously analyze the security of the improved protocol.

3. Establish formal and precise security requirements that should be satisfied by a network time-synchronization protocol, and prove that these are sufficient for the security of other protocols that rely on time.

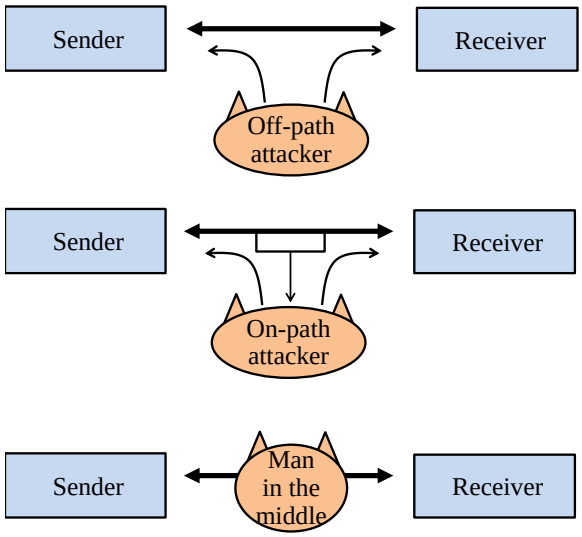

Figure 1·3: Threat models

1.6 Our Contributions

The following chapters in this thesis are self-contained with their own introduction, motivation, results and conclusion sections. This section gives an overview of the results in these chapters, and discuss how they contribute incrementally to achieve the goals of this thesis.

1.6.1 Attacking and fixing different modes of NTP (unauthenticated and authenticated) Chapters 2, 3, 5

Threat Models. We consider the following categories of network attacks on NTP (See Figure 1·3).

1. **Off-path attacks.** An off-path attacker cannot eavesdrop on the NTP traffic of its targets, but can spoof IP packets *i.e.,* send packets with a bogus source IP address. This threat model captures 'remote attacks' launched by arbitrary IPs that do not occupy a privileged position on the communication path between the NTP client and server. Off-path attacks are essentially the weakest (and therefore the scariest) threat model that one could consider for a networking

protocol.

2. **On-path attacks.** An on-path attacker occupies a privileged position on the communication path between NTP client and one of its servers, or hijacks (with *e.g.,* DNS (Kaminsky, 2008), (Herzberg and Shulman, 2013) or BGP (Goldberg, 2014), (d'Itri, 2015; Peterson, 2013)) traffic to the server. An on-path attacker can eavesdrop, inject, spoof, and replay packets, but *cannot* drop, delay, or tamper with legitimate traffic. An on-path attacker eavesdrops on a copy of the target's traffic, so it need not disrupt live network traffic, or even operate at line rate. For this reason, on-path attacks are commonly seen in the wild, disrupting TCP (Weaver et al., 2009), DNS (Duan et al., 2012), BitTorrent (Weaver et al., 2009), or censoring web content (Clayton et al., 2006).

3. **In-path/MiTM attacks.** An in-path attacker or more traditionally a Man-in-The-Middle (MiTM) can eavesdrop, inject, spoof, replay and *can additionally* delay, drop, or tamper with the legitimate traffic.

Our Attacks. We demonstrate several attacks on NTP client. Each of these attacks (a) exploits distinct *flaws* in the protocol or its implementation, (b) requires different *resources* in terms of number of spoofed packets, time to accomplish the attack, and position of the attacker on network path (off-, on- or in-path), (c) requires different *fixes* to the protocol design or implementation.

Impact. Overall our attacks and Internet measurements expose the fragility of NTP ecosystem. We show that the attacks are possible due to the following:

1. flaws in NTP protocol design and underspecified RFCs.

2. despite conflicting security-requirements, RFC 5905 suggests that implementations process different NTP modes[4] in the same codepath. Also, this lack of

[4]Other than the typical *client-server* mode, NTP has several other modes of operation, *e.g.,*

modularity in code-design makes it challenging to modify code without inadvertently (re)introducing more vulnerabilities.[5]

3. lack of secure protocol for cryptographically-authenticating NTP.

We make several recommendations to fix NTP's existing reference implementation that were incorporated in the ntpd software thus making it more robust and secure[6]. Most importantly, however, our work motivates the community to work towards better protocol design (Schiff et al., 2018), (roughtime, 2015), efforts to make existing implementations more robust and secure ((Stenn, 2015b), (NTPsec, 2017)), and proposals for cryptographically-authenticating NTP (Dowling et al., 2016), (Mkacher et al., 2018) and (Franke et al., 2018), (Malhotra and Goldberg, 2019).

Following is a brief overview of some of our attacks on NTP (more details in respective chapters).

Time-shifting attacks.[7] In time-shifting attacks, a network attacker can maliciously *alter time* on client systems.

1. Time steps by on-path and off-path attackers. Our on-path attacks on NTP's client-server mode exploit client's behavior upon initialization, and the fact that an on-path attacker can easily determine exactly when an ntpd client is initializing. We show "small-step-big-step" attack that stealthily shifts client clocks when clients are unlikely to notice. One might suppose that preconfiguring an NTP client with multiple servers could prevent this attack. However, we show that an attacker need only intercept traffic to *one* server queried by a client. We then show how an off-path

broadcast mode, where a set of clients are pre-configured to listen to a server that broadcasts timing information, and *symmetric* mode where peers exchange time information typically on the same level. There is also an *interleaved* mode that enables NTP servers to provide their clients and peers with more accurate timestamps that are available only after transmitting NTP packets.

[5]e.g., our "Zero-0rigin timestamp" attack was fixed in ntpv4.2.8p6. However, subsequent timestamp validation checks introduced a regression in the handling of some Zero origin timestamp checks.

[6]We secured several Common Vulnerabilities and Exposure (CVE, 2019) in the process.

[7]Some attacks were discovered by members of Cisco security team ASIG (specifically Matt Van Gundy, Jonathan Gardner) while the author was a visiting research fellow at Cisco, Knoxville.

attacker, that uses spoofed rate-limiting *Kiss-o'-Death (KoD)* packets, forces clients to synchronize to malfunctioning servers that provide incorrect time.

2. Leaky-origin timestamp attack. In this attack we leverage our observation that NTP's control query interface is *not specified* in the latest RFC 5905 and can be exploited remotely as a side-channel to *leak* information about NTP client's internal timing state variables. We show that an off-path attacker can maliciously shift time on a target client by querying for client's secret state variables and thus bypassing the non-cryptographic authentication checks.

3. Interleaved pivot attack. In this attack we exploit the fact that in NTP's reference implementation, client-server mode shares the same codepath as NTP's interleaved mode. First, the attacker spoofs a single packet, for an NTP client operating in client-server mode, that tricks the target into thinking that he is in interleaved mode. There is no codepath that allows the client to exit interleaved mode. The target then rejects all subsequent legitimate client-server mode packets. This is a DoS attack. We further leverage NTP's leaky control queries to convert this DoS attack to an on-path timeshifting attack.

4. Zero-0rigin timestamp attack. This is among the strongest timeshifting attacks on NTP's client-server mode and follows directly from RFC 5905 specification. In this attack we exploit the fact that NTP's client-server mode shares the same codepath as NTP's symmetric mode. The off-path attacker bypasses non-cryptographic authentication check by spoofing server response packets (with malicious timing information) for the client with their nonce set to a predictable value "zero" (a legitimate requirement for symmetric mode).

5. Fragmentation attack. We show how NTP's interaction with lower layer protocols (UDP, ICMP, IP) can be exploited in an off-path IPv4 fragmentation attack that can dramatically shift time on a victim client. We explain why NTP's clock discipline

algorithms force our attack to craft a *stream* of self-consistent packets (rather than just one packet, as in the attacks of *e.g.,* (Kaminsky, 2008), (Herzberg and Shulman, 2013)), and demonstrate that this can be done with a proof-of-concept implementation.

6. Replay attack. This attack exploits the weakness in NTP's broadcast mode that the check for replay protection does not actually prevent replay in general; it only prevents replay for the most recent packet. We show that an on-path attacker can indefinitely stick a cryptographically-authenticated broadcast client to a specific time.

Denial-of-Service (DoS) attacks. In DoS attacks, a network attacker prevents the victim client system from ever updating its clock.

1. DoS attack-1. RFC 5905 requires that the server sends a client a KoD packet asking the client to reduce its polling interval if the client queries it too many times within a specified time interval. RFC 5905 also requires that the client MUST immediately reduce its polling interval to that sent by the server and continue to reduce it each time it receives this packet. But RFC 5905 *does not explicitly* require the client to check for authentication for KoD packet. We show how an off-path attacker can essentially disable NTP for any victim. Our attacker need only spoof a single KoD packet with large polling interval from each of the client's preconfigured servers, and send this packet once every 36 hours. The client stops querying its servers and no longer updates its local clock. An off-path attacker that uses standard networking scanning tools (e.g., zmap (Durumeric et al., 2013)) to spoof KoD packets can launch this attack on most NTP clients in the Internet within a few hours, essentially disabling NTP in the Internet.

2. DoS attack-2. This attack exploits a flaw in NTP's broadcast mode[8] of operation that an off-path attacker can easily cause an error by sending a packet with bad cryp-

[8]In broadcast mode a set of NTP clients are pre-configured to listen to a server that broadcasts timing information.

tographic authentication (e.g., wrong or mismatched key, incorrect message digest, etc.). We show a DoS attack where an off-path attacker sends one such error-causing packet per query interval, so that a cryptographically-authenticated broadcast client immediately tears down its association with the server. This way, the client never collects enough good NTP responses to allow its clock discipline algorithms to update its local clock. This attack also applies to all other NTP modes that are 'ephemeral'[9] or 'preemptable' (including manycast, pool, etc). This attack implies that even if cryptographic-authentication is used, an off-path attacker can still spoof packets that will deny NTP service to a broadcast client.

1.6.2 Leveraging NTP to attack DNS (Chapter 4)

Progressing towards the goal of analyzing the impact of security vulnerabilities in network timing protocols (including but not limited to NTP) on other protocols, we demonstrate this on Domain Name System (DNS) and the DNS Security Extensions (DNSSEC) protocol.

How does DNS rely on time? DNS relies on caching to provide enhanced performance and improved reliability in the face of network failures. DNS *caches* rely on an absolute notion of time (e.g., "August 8, 2019 at 11:00am") to determine how long DNS records can be cached (i.e., their Time To Live (TTL)) and to determine the validity interval of DNSSEC signatures. Following our two-step approach described above in subsection 1.6.1, we find that while TTL is a relative time value (e.g., "2 hours from the time the DNS query was sent"), the RFCs *do not* clearly specify how the cache should determine that the TTL has elapsed. We then checked popular caching resolver implementations (Unbound, 2018), (Bind, 2018), (Powerdns, 2018), (Dnsmasq, 2018), (knot, 2018) (before v1.5.1) to see how their caches were implementing time. We found that these implementations all mark the end of validity of

[9] An ephemeral association is mobilized upon arrival of a packet and exists until error or timeout.

the cached object by translating the relative time values in the TTL into absolute time values by adding an offset equal to the TTL to the current system time. DNS resolver implementations, however, do not come with a predefined mechanism for getting absolute time. So the best that they can do is to rely on system time (which represents some form of absolute time) from the underlying OS to get these absolute time values.

Previous work on DNS attacks. Several works study DNS spoofing and cache poisoning attacks and propose potential solutions (Kaminsky, 2008), (Herzberg and Shulman, 2012a), (Klein et al., 2017), (Herzberg and Shulman, 2012b), (Schomp et al., 2014), (Yuan et al., 2006). These attacks focus on *how* the DNS protocol can be exploited to poison DNS resolver caches. By contrast our attacks show how to pivot from vulnerabilities in absolute time (*e.g.,* in NTP), to attacks that flush/stick the DNS cache.

Impact. Our attacks indicate that DNS caches should lessen their reliance on absolute time. In fact, we observe that setting TTL as relative time value is a feature of the DNS protocol. So there is no fundamental change required to the DNS protocol to deal with our attacks. We therefore recommend that DNS caching resolvers use relative time rather than absolute time. Specifically, we suggest using the OS's raw time, which is monotonically increasing and not subject to adjustment by external sources (and thus immune to the network attacks on time.) We have implemented our recommendations as part of the popular open source validating, recursive, and caching DNS resolver product from NLnet Labs.

Our Attacks. We leverage the above mentioned observations to investigate a largely overlooked and important threat in DNS: the impact of security vulnerabilities introduced because resolver's caches are dependent on absolute time, which is obtained from the system time, which is often obtained via network timing protocols (e.g.,

NTP), which are vulnerable to attacks. We show how to pivot from network attacks on absolute time (NTP) to attacks on DNS caching. Specifically, we present and discuss the implications of following attacks:

1. Cache-expiration attacks. When time is shifted forwards, the DNS cached responses expire sooner than expected, effectively flushing the cache.

2. Cache-sticking attacks. When time is shifted backwards, the cached responses stick in the cache for longer than intended.

We show how these attacks can be used to harm DNS performance (introducing latency into DNS responses) and DNS availability (increasing the risk of denial-of-service). We also discuss how they can be used to aid for fast-fluxing (Holz et al., 2008), (Honeynet, 2018), cache poisoning and other well-known threats to the DNS. We use network measurements to identify a significant attack surface for these DNS cache attacks in the Internet.

1.6.3 New provably-secure NTP (Chapter 5)

Having demonstrated the fragility of NTP and its impact on other protocols, our next goal is to go beyond attacks and patches and *propose* a more robust protocol and *prove the security* of new protocol. In Chapter 5 we propose a new backwards-compatible protocol for client-server mode that preserves the semantics of the timestamps in NTP packets. We then leverage ideas from the universal composability framework (UC) (Canetti, 2001) to develop a cryptographic model for network attacks on NTP's datagram protocol.

Impact. Our network protocol analysis introduces new ways of reasoning about network attacks on time synchronization protocols. In this work, our focus is in securing the core datagram protocol used by NTP. To the best of our knowledge, the security of the core NTP datagram protocol has never previously been analyzed.

We also prove the security of authenticated NTP where we assume authenticated communication channels. As such, our design and analysis of secure client-server protocols complements recent proposals of protocols for establishing authenticated communication channels *i.e.,* protocols for distributing keys and performing the MAC (Haberman and Mills, 2010), (Dowling et al., 2016), (Franke et al., 2018), (Itkin and Wool, 2016).

Threat model. Our model is inspired by prior cryptographic work that designs synchronous protocols with guaranteed packet delivery (Katz et al., 2013a),(Achenbach et al., 2015). However, we consciously omit to model the more powerful MiTM who can drop, modify, or delay packets.

Why is MiTM too strong a threat model for NTP? We do *not* consider in-path attacks. This is because an MiTM can always prevent time synchronization by dropping packets. Moreover, an MiTM can also bias time synchronization by delaying packets (Mizrahi, 2012a; Mizrahi, 2012b). This follows because time synchronization protocols use information about the delay on the network path in order to accurately synchronize clocks (Section 5.2). A client cannot distinguish the delay on the forward path (from client to server) from the delay on the reverse path (from server to client). As such, the client simply takes the total round trip time δ (forward path + reverse path), and assumes that delays on each path are symmetric. The MiTM can exploit this by making delays asymmetric (*e.g.,* causing the delay on the forward path to be much longer than delay on the reverse path), thus biasing time synchronization.

We assume instead that the network delivers all packets sent between the honest parties. We also assume that the network does not validate the source IP in the packets it transits, so that the attacker can *spoof* packets. Honest parties experience a delay before their packets are delivered, but the attacker can win every race condition. We require that honest parties have sufficient time to process every packet received;

put another way, attacks that flood an honest party with packets in order to deny service are out of scope.

Security guarantees. We use this model to prove that our proposed protocol *correctly synchronizes time* in the face of both (1) *off-path attackers* when NTP is unauthenticated, and (2) *on-path attackers* when NTP packets are authenticated with a cryptographically secure MAC (Refer to the corresponding Theorems in Chapter 5.) We also use our model to prove similar results about a different protocol that is used by chronyd (chronyd, 2015) and openNTPD (openNTPD, 2012) (two alternate implementations of NTP). The chronyd/openNTPD protocol is secure, but unlike our protocol, does not preserve the semantics of packet timestamps.

1.6.4 A Universally-Composable Treatment of Network Time (Chapter 6)

Up until now, our work has specifically focused on the design and analysis of the most dominant protocol for time synchronization in the Internet NTP. We observe that meanwhile we have many candidates for secure network time synchronization protocols like (Haberman and Mills, 2010), (Sibold et al., 2015), (Dowling et al., 2016), (Itkin and Wool, 2016), (Franke et al., 2018). We also have several frameworks (Mizrahi, 2012b), (Dowling et al., 2016), (Itkin and Wool, 2016), (Malhotra et al., 2017) that aim to define and analyze the security requirements of time synchronization protocols. Additionally there exist many works that propose ways to model time (either real, global, or relative) within network protocols, and even within security protocols (Kalai et al., 2005), (Katz et al., 2013b), (Canetti, 2013), (Backes et al., 2014) and (Vajda, 2016) to analyze their security.

However none of these works rigorously capture the security guarantees from a network time synchronization protocol that provably suffice for security-sensitive applications that require time— for instance for guaranteeing the validity of cryptographic certificates in a way that, in turn, will guarantee authenticated and secure

communication.

So the next goal of this thesis is to establish formal and precise security requirements that should be satisfied by a network time synchronization protocol, and prove that these are sufficient for the security of other time-reliant applications – specifically, cryptographic certificates with revocation and expiration times.

Our Contributions. We choose the Universally Composable (UC) security framework as a basis for our formalism because of its two specific properties that are important for our modeling:

1. Secure composition: UC framework provides a general mechanism for specifying security properties of cryptographic protocols in a way that facilitates composing protocols together, and in particular guarantees that composition of secure components results in overall security of the composed protocol. This allows for relatively clear and modular treatment of the use of time in security-sensitive systems.

2. Seamless integration of real time within existing cryptographic tools and primitives: UC framework is geared towards analyzing the security of cryptographic protocols, which facilitates incorporating the results in this work with existing analytical results for cryptographic protocols.

Both of these properties are crucial towards our goal of realizing time-sensitive cryptographic primitives like the Public Key Infrastructure (PKI). Specifically, we propose formal abstractions of secure network time, and show that:

- Our abstractions of network-time suffice for securely incorporating expiration times in certificates, as well as freshness guarantees for public certificate lists, in a way that guarantees PKI-based secure communication *even in face of an*

adversary who tries to subvert the measurement of time and at the same time corrupts revoked and expired certificates.

- Our abstractions are realizable by simple protocols that mimic the behavior of authenticated NTP.

Specifically, we build upon an existing analytical work by Canetti et al. that asserts, within the UC framework, the security of authentication and key exchange protocols that are based on global PKI (Canetti et al., 2016). We incorporate our analysis of timing consensus achieved via network time into the UC analysis of a global PKI. The combined analysis extends the security guarantees provided by (Canetti et al., 2016) to the case of revocable and expirable certificates.

Impact. Our methodology of incorporating network time into existing UC protocols and functionalities is quite generic. Hence, our work paves the way toward instantiating time consensus and reaping its security benefits within other UC formalisms in a seamless fashion. An important lesson learned from this modeling is that real-life certificate revocation lists and online certificate status protocol (OCSP) requests must continue to answer requests about revoked or expired certificates during the interval $[t^*, t^* + \Sigma]$ because clients may not be able to adjudicate them correctly on their own before this time. Here, Σ denotes the maximum shift from real time expected by Theorems 4 and 5 for all clients on the Internet. After this interval, the adversary cannot convince any clients of the validity of a revoked or expired certificate via network manipulation, so the CA may forget about its existence.

Chapter 2

Attacking the Network Time Protocol

2.1 Introduction

NTP (Mills et al., 2010) is one of the Internet's oldest protocols, designed to synchronize time between computer systems communicating over unreliable variable-latency network paths. NTP has recently received some attention from security researchers due to software-implementation flaws (National Vulnerability Database, 2014), (Röttger, 2015), and its potential to act as an amplifier for distributed denial of service (DDoS) attacks (Czyz et al., 2014a), (Stenn, 2015d). However, the community still lacks visibility into the robustness of the NTP ecosystem itself, as well as the integrity of the timing information transmitted by NTP. These issues are particularly important because time is a fundamental building block for computing applications, and is heavily utilized by many cryptographic protocols.

NTP most commonly operates in an hierarchical client-server fashion. Clients send queries to solicit timing information from a set of preconfigured servers that usually remain static over time. At infrequent intervals, a client adaptively select a single server to which it synchronizes its local clock; selection decisions are made by a series of complex *clock discipline* algorithms that are (sometimes incompletely) specified in RFC 5905 (Mills et al., 2010), and have evolved in small but important ways across different versions of *ntpd*, the NTP reference implementation. While NTP supports both symmetric and asymmetric cryptographic authentication (Haberman and Mills, 2010), in practice, these modes of operation are rarely used (Section 2.3).

Our goal is therefore to explore attacks on unauthenticated NTP that are possible *within* the NTP protocol specification (Mills et al., 2010). We consider both (1) *on-path attacks*, where the attacker occupies a privileged position on the path between NTP client and one of its servers, or hijacks (with *e.g.,* DNS (Kaminsky, 2008), (Herzberg and Shulman, 2013) or BGP (Goldberg, 2014), (d'Itri, 2015), (Peterson, 2013)) traffic to the server, and (2) *off-path attacks*, where the attacker can be anywhere on the network and does not observe the traffic between client and any of its servers. This chapter considers the following.

Implications (Section 2.2). We consider a few implications of attacks on NTP, highlighting protocols and applications whose correctness and security relies on accurate time.

On-path time-shifting attacks (Section 2.4). We discuss how an on-path attacker can shift time on victim clients by hours or even years. Our attacks exploit NTP's behavior upon initialization, and the fact than an on-path attacker can trivially determine when an ntpd client is initializing. We also present a "small-step-big-step" attack (CVE-2015-5300) that stealthily shifts clocks when clients are unlikely to notice.

Off-path DoS attacks (Section 2.5.3). We show how an off-path attacker can disable NTP at a victim client by exploiting NTP's rate-limiting mechanism, the *Kiss-o'-Death (KoD)* packet.

1. DoS by Spoofed Kiss-o'-Death (CVE-2015-7704). We show how a single attacking machine can disable NTP on most of clients in the Internet. We find that ntpd versions earlier than 4.2.8p4 allow an off-path attacker to trivially spoof a KoD packet for each of the client's preconfigured servers; upon receipt of the spoofed KoD, the client stops querying its servers and stops updating its clock. Because the attacker only sends a few KoD packets per client, standard network scanning tools (nmap, zmap (Durumeric et al., 2013)) can be used to quickly launch this attack, in bulk,

on all ntpd clients in the Internet. This vulnerability was patched in ntpd v4.2.8p4 following our work.

2. DoS by Priming the Pump (CVE-2015-7705). Even if KoD packets can no longer be trivially spoofed, an off-path attacker can still disable NTP at a victim client; this attack, however, requires the attacker to expend more resources (*i.e.,* send more packets). Our off-path attacker sends the servers a high volume of queries that are spoofed to look like they come from the client. The servers then respond to any subsequent queries from the client with a valid KoD, and once again, the client stops querying its servers, and stops updating its local clock. Our recommended mitigations are in Section 2.5.7.

Off-path time-shifting attacks. Next, we consider off-path attackers that step time on victim NTP clients:

1. Pinning to bad timekeepers (Section 2.5.4). We first consider an off-path attacker that uses spoofed KoD packets to force clients to synchronize to malfunctioning servers that provide incorrect time; we find that NTP is pretty good at preventing this type of attack, although it succeeds in certain situations.

2. Fragmentation attack (Section 2.6). Then we show how NTP's interaction with lower layer protocols (ICMP, IPv4) can be exploited in a new off-path IPv4 fragmentation attack that shifts time on a victim client. We explain why NTP's clock discipline algorithms require our attack to craft a *stream* of self-consistent packets (rather than just one packet, as in (Kaminsky, 2008), (Herzberg and Shulman, 2013)), and demonstrate its feasibility with a proof-of-concept implementation. This attack, which has a small but non-negligible attack surface, exploits certain IPv4 fragmentation policies used by the server and client operating systems (Section 2.6.5), rather than specific issues with NTP.

Network measurements (Sections 2.3.2, 2.5.6, 2.6.7, 2.6.8). The last measurement

studies of the NTP ecosystem were conducted in 1999 (Minar, 1999) and 2006 (Murta et al., 2006), while a more recent study (Czyz et al., 2014a) focused on NTP DoS amplification attacks. We study the integrity of the NTP ecosystem using data from the openNTPproject (Mauch, 2015), and new network-wide scans (Section 2.3.2).

We identify bad timekeepers that could be exploited by off-path attacker (Section 2.5.6), and servers that are vulnerable to our fragmentation attack (Sections 2.6.7-2.6.8).

Recommendations and disclosure. We began disclosing these results on August 20, 2015. The Network Time Foundation, NTPsec, Redhat's security team, and Cisco quickly patched their NTP implementations to prevent trivial spoofing of the KoD packet (CVE-2015-7704). We also worked with the openNTPproject to provide a resource that operators can use to measure their servers' vulnerability to our fragmentation attacks.[1] Our recommendations for hardening NTP are in Sections 2.4.3, 2.5.7, 2.6.9 and summarized in Section 2.8.

2.2 Why time matters: Implications of attacks on NTP

NTP lurks in the background of many systems; when NTP fails on the system, multiple applications on the system can fail, all at the same time. Such failures have happened. On November 19, 2012 (Bicknell, 2012), for example, two important NTP (stratum 1) servers, `tick.usno.navy.mil` and `tock.usno.navy.mil`, went back in time by about 12 years, causing outages at a variety of devices including Active Directory (AD) authentication servers, PBXs and routers (Morowczynski, 2012). Exploits of individual NTP clients also serve as a building block for other attacks, as summarized in Table 2.1. Consider the following:

TLS Certificates. Several authors (Mills, 2011, pg 33, pg 183) (Klein, 2013; Selvi, 2015) have observed that NTP could be used to undermine the security of TLS cer-

[1] `http://www.cs.bu.edu/~goldbe/NTPattack.html`

tificates, which are used to establish secure encrypted and authenticated connection. An NTP attacker that sends a client back in time could cause the host to accept certificates that the attacker fraudulently issued (that allow the attacker to decrypt the connection), and have since been revoked[2]. (For example, the client can be rolled back to March 2011, when a compromise at Comodo (Comodo, 2011) allowed hackers to issue fraudulent certificates for domains including `*.google.com`, or back to mid-2014, when $> 100K$ certificates were revoked due to heartbleed (Zhang et al., 2014).) Alternatively, an attacker can send the client back to a time when a certificate for a cryptographically-weak key was still valid. (For example, to 2012 when (Heninger et al., 2012) exploited entropy problems in key generation to obtain private keys for 0.50% of the Internet's TLS hosts, or to 2008, when a bug in Debian OpenSSL caused thousands of certificates to be issued for keys with only 15-17 bits of entropy (Eckersley and Burns, 2010).) Moreover, most browsers today accept (non-root) certificates for 1024-bit RSA keys, even though sources speculate that they can be cracked by well-funded adversaries (Barker and Roginsky, 2011); thus, even a domain that revokes its old 1024-bit RSA certificates (or lets them expire) is vulnerable to cryptanalytic attacks when its clients are rolled back to a time when these certificates were valid. Some of these attacks were demonstrated by Selvi (Selvi, 2015).

HSTS. Selvi (Selvi, 2014) suggests using NTP to attack HTTP Strict Transport Security (HSTS). HSTS is used by a webserver to tell its client that all future connections should use HTTPS. The HSTS request, however, comes with an expiry time (one year is recommended (OWASP, 2015)) after which point the client need no longer

[2]The attacker must also circumvent certificate revocation mechanisms, but several authors (Mutton, 2014), (Kiyawat, 2014), (Langley, 2014) point out that this is relatively easy to do in various settings. For instance, several major browsers rely on OCSP (Santesson et al., 2013) to check if a certificate was revoked, and default to "soft-fail", *i.e.,* accepting the certificate as valid, when they cannot connect to the OCSP server. NTP-based cache-flushing could also be useful for this purpose, by causing the client to 'forget' any old certificate revocation lists (CRLs) that it may have seen in the past; see also our discussion of routing attacks.

Table 2.1: Attacking various applications with NTP.

To attack...	change time by ...	To attack...	change time by ...
TLS Certificates	years	Routing (RPKI)	days
HSTS	a year	Bitcoin	hours
DNSSEC	months	API authentication	minutes
Kerberos	minutes	DNS Caches	days

use an encrypted connection; thus, an NTP attacker that sends the client a year into the future can effectively disable HSTS, allowing for downgrade attacks (*e.g.,* 'SSL Stripping' (Marlinspike, 2009)) to non-encrypted connections.

DNSSEC. DNSSEC (Arends et al., 2005a) provides cryptographic authentication of the Domain Name System (DNS) data. NTP can be used to attack a DNS resolver that performs 'strict' DNSSEC validation, *i.e.,* fails to return responses to queries that fail cryptographic DNSSEC validation. An NTP attack that sends a resolver forward in time will cause all timestamps on DNSSEC cryptographic keys and signatures to expire (the recommended lifetime for zone-signing keys in DNSSEC is 1 month (Kolkman et al., 2012)); the resolver and all its clients thus lose connectivity to any domain secured with DNSSEC. Alternatively, an NTP attack that sends a resolver back in time allows for DNSSEC replay attacks; the attacker, for example, roll to a time in which a certain DNSSEC record for a domain name did not exist, causing the resolver to lose connectivity to that domain. Since the recommended lifetime for DNSSEC signatures is no more than 30 days (Kolkman et al., 2012), this attack would need to send the resolver back in time by a month (or more, if the time in which the DNSSEC record did not exist was further in the past). Going back in time also allows an attacker to forge DNSSEC responses using an old DNSSEC key that is compromised or cryptographically weak. Indeed, DNSSEC resolvers still accept 1024-bit RSA signatures (Kolkman et al., 2012), so even if zone decides to protect itself by upgrading to a stronger key, an NTP attack could still roll back its clients to a time when the old 1024-bit key was still valid. (Indeed, 1024-bit RSA

was the most popular DNSSEC key in 2014, although we hope it won't be in the future (Herzberg et al., 2014).)

Cache-flushing attacks. NTP can be used to flush caches. The DNS, or example, uses caching to minimize the number of DNS queries a resolver makes to a public nameserver, thus limiting network traffic. DNS cache entries typically live for around 24 hours, so rolling a resolver forward in time by a day would cause most of its cache entries to expire (Mills, 2011), (Klein, 2013). A widespread NTP failure (like the one in November 2012) could cause multiple resolvers to flush their caches all at once, simultaneously flooding the network with DNS queries.

Interdomain routing. NTP can be used to exploit the Resource Public Key Infrastructure (RPKI) (Lepinski and Kent, 2012), a new infrastructure for securing routing with BGP. The RPKI uses Route Origin Authorizations (ROAs) to cryptographically authenticate the allocation of IP address blocks to networks. ROAs prevent hijackers from announcing routes to IP addresses that are not allocated to their networks. If a valid ROA is missing, a 'relying party' (that relies on the RPKI to make routing decisions) can lose connectivity to the IPs in the missing ROA.[3] As such, relying parties must always download a complete set of valid ROAs; to do this, they verify that they have downloaded all the files listed in cryptographically-signed 'manifest' files. To prevent the relying party from rolling back to a stale manifest that might be missing a ROA, manifests have monotonically-increasing 'manifest-numbers', and typically expire within a day (Heilman et al., 2014). NTP attacks, however, can first roll the relying party forward in time, flushing its cache and causing it to 'forget' its current manifest-number, and then roll the relying party back in time, so that it accepts a stale manifest as valid.

Bitcoin. Bitcoin is a digital currency that allows a decentralized network of nodes

[3]See (Cooper et al., 2013, Side Effect 6): the relying party loses connectivity if it uses 'drop invalid' routing policy (Cooper et al., 2013, Sec. 5), and the missing ROA has 'covering ROA'.

to arrive at a consensus on a distributed public ledger of transactions, *aka* "the blockchain". The blockchain consists of timestamped "blocks"; bitcoin nodes use computational proofs-of-work to add blocks to the blockchain. Because blocks should be added to the blockchain according to their validity interval (about 2 hours), an NTP attacker can trick a victim into rejecting a legitimate block, or into wasting computational power on a stale block (Corbixgwelt, 2011).

Authentication. Various services (*e.g.,* (Amazon, 2015), (DropBox, 2015)) expose APIs that require authentication each time an application queries them. To prevent replay attacks, queries require a timestamp that is within some short window of the server's local time, see *e.g.,* (Hammer-Lahav, 2010, Sec 3.3); Amazon S3, for example, uses a 15-minute window. An NTP attacker can also launch replay attacks on Kerberos, which requires clients to present tickets which have been timestamped within minutes (Kohl and Neuman, 1993).

2.3 The NTP Ecosystem

We start with background on the NTP protocol, and use a measurement study to discuss its structure and topology. NTP has evolved in more fluid fashion than other core Internet protocols like DNS or BGP. While NTP is described in RFC 5905 (Mills et al., 2010), practically speaking, the protocol is determined by the NTP reference implementation *ntpd*, which has changed frequently over the last decades (Stenn, 2015d). (For example, *root distance* Λ (equation (2.4)) is a fundamental NTP parameter, but is defined differently in RFC 5905 (Mills et al., 2010, Appendix A.5.5.2), ntpd v4.2.6 (the second most popular version of ntpd we found in the wild) and ntpd v4.2.8 (the latest version).)

2.3.1 Background: The NTP Protocol.

NTP most commonly operates in an hierarchical client-server fashion.[4] Clients send queries to solicit timing information from a set of servers. This set of servers is manually configured before the client initializes and remains static over time. In general, the ntpd client can be configured with up to 10 servers.[5] Online resources suggest configuring anywhere from three to five servers (Knowles, 2004), and certain OSes (*e.g.,* MAC OS X 10.9.5) default to installing ntpd with exactly one server (*i.e.,* `time.apple.com`). At the root of the NTP hierarchy are *stratum 1* NTP servers, that provide timing information to stratum 2 client systems. Stratum 2 systems provide time to stratum 3 systems, and so on, until stratum 15. Stratums 0 and 16 indicate that a system is unsynchronized. NTP servers with low stratum often provide time to the Internet at large (*e.g.,* `pool.ntp.org`, `tick.usno.navy.mil`); our organization, for example, has stratum 2 servers that provide time to internal stratum 3 machines, and take time from public stratum 1 servers.

Client/server communications. An NTP client and server periodically exchange a pair of messages; the client sends the server a *mode 3* NTP query and the server responds with a *mode 4* NTP response. This two-message exchange uses the IPv4 packet shown in Figure 2·1, and induces the following four important timestamps on the mode 4 response:

T_1 *Origin timestamp.* Client's system time when client sent mode 3 query.

T_2 *Receive timestamp.* Server's system time when server received mode 3 query.

T_3 *Transmit timestamp.* Server's system time when server sent mode 4 response.

[4]NTP also supports several less popular modes including *broadcast*, where a set of clients listen to a server that broadcasts timing information, and *symmetric peering*, where servers (typically at the same stratum) exchange time information. We only consider client-server mode.

[5]For example, when installing NTP in 14.04.1-Ubuntu in July 2015, the OS defaulted to installing ntpd v4.2.6 with a five preconfigured servers.

T_4 *Destination timestamp.* Client's system time when client received mode 4 response. (Not in packet.)

The round-trip *delay* δ during the exchange is therefore:

$$\delta = (T_4 - T_1) - (T_3 - T_2) \tag{2.1}$$

Offset θ quantifies the time shift between a client's clock and a server's clock. Assume that delays on the forward (client→server) and reverse (server→client) network paths are symmetric and equal to $\frac{\delta}{2}$. Then, the gap between the server and client clock is $T_2 - (T_1 + \frac{\delta}{2})$ for the mode 3 query, and $T_3 - (T_4 - \frac{\delta}{2})$ for the mode 4 response. Averaging these two quantities gives the offset:

$$\theta = \tfrac{1}{2}\left((T_2 - T_1) + (T_3 - T_4)\right) \tag{2.2}$$

An NTP client adaptively and infrequently selects a *single* server (from its set of pre-configured servers) from which it will take time. The IPv4 address of the selected server is recorded in the *reference ID* field of every NTP packet a system sends, and the *reference timestamp* field records the last time it synchronized to its reference ID. Notice that this means that any client querying a server S_2 can identify exactly which IPv4 NTP server S_1 the server S_2 has used for synchronization.[6]

Infrequent clock updates. NTP infrequently updates a client's clock because (1) a client and server must exchange between eight to hundreds of messages before the client's *clock discipline algorithms* synchronizes it to the server (Mills et al., 2010, Sec. 10-12) (see also Section 2.5.5), and (2) messages are exchanged at infrequent *polling intervals* (on the order of minutes) that are adaptively chosen by a randomized *poll process* (Mills et al., 2010, Sec. 13).

[6]128-bit IPv6 addresses are first hashed and then truncated to before recorded in the 32-bit reference ID field (Mills et al., 2010, pg 22). Thus, one would need a dictionary attack to identify an IPv6 server.

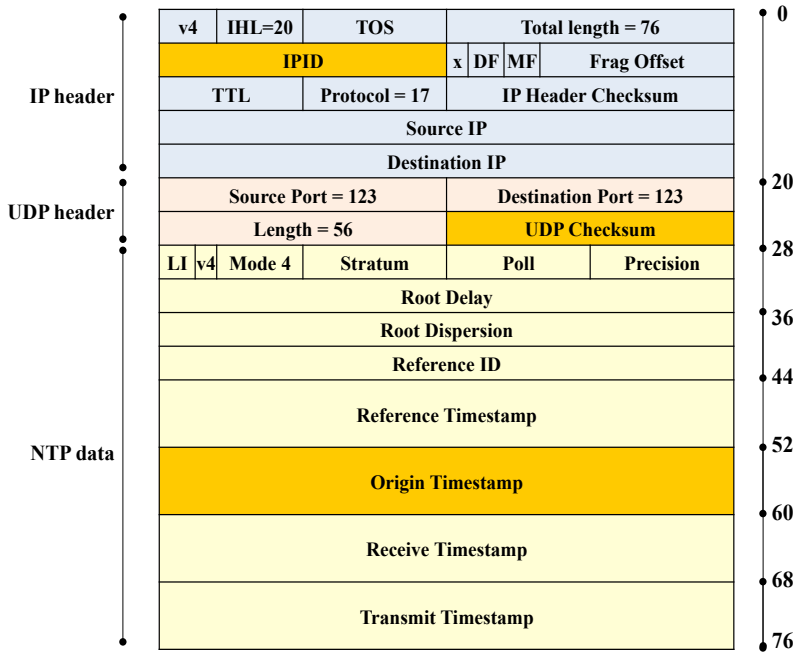

Figure 2·1: Mode 4 NTP Packet, highlighting nonces and checksums.

Authentication. How does the client know that she is talking to its real NTP server and not to an attacker? While NTPv4 supports both symmetric and asymmetric cryptographic authentication, this is rarely used in practice. Symmetric cryptographic authentication appends an MD5 hash keyed with symmetric key k of the NTP packet contents m as MD5($k||m$) (Mills, 2011, pg 264) to the NTP packet in Figure 2·1.[7] The symmetric key must be pre-configured manually, which makes this solution quite cumbersome for public servers that must accept queries from arbitrary clients. (NIST operates important public stratum 1 servers and distributes symmetric keys only to users that register, once per year, via US mail or facsimile (NIST, 2010); the US Naval Office does something similar (USNO, 2015).) Asymmetric cryptographic au-

[7]MD5($k||m$) is intended to provide be a cryptographic message authentication code (MAC), but the use of MD5 in this manner is not a provably secure MAC and is also vulnerable to length-extension attacks; HMAC should be used instead (Bellare et al., 1996). Moreover, MD5 has been depreciated in favor of more secure hash functions like SHA-256 (Turner and Chen, 2011).

thentication is provided by the Autokey protocol, described in RFC 5906 (Haberman and Mills, 2010). RFC 5906 is not a standards-track document (it is classified as 'Informational'), NTP clients do not request Autokey associations by default (Autokey, 2012), and many public NTP servers do not support Autokey (*e.g.,* the NIST timeservers (NIST, 2010), many servers in `pool.ntp.org`). In fact, a lead developer of the ntpd client wrote in 2015 (Stenn, 2015a): "Nobody should be using autokey. Or from the other direction, if you are using autokey you should stop using it." For the remainder of this chapter, we shall assume that NTP messages are unauthenticated.

2.3.2 Measuring the NTP ecosystem.

We briefly discuss the status of today's NTP ecosystem. Our measurement study starts by discovering IP addresses of NTP servers in the wild. We ran a zmap (Durumeric et al., 2013) scan of the IPv4 address space using mode 3 NTP queries on April 12-22, 2015, obtaining mode 4 responses from 10,110,131 IPs.[8] We augmented our data with openNTPproject (Mauch, 2015) data from January-May 2015, which runs weekly scans to determine which IPs respond to NTP control queries. (These scans are designed to identify potential DDoS amplifiers that send large packets in response to short control queries (Czyz et al., 2014a).) The openNTPproject logs responses to NTP *read variable (rv)* control queries. *rv* responses provide a trove of useful information including: the server's OS (also useful for OS fingerprinting!), its ntpd version, its reference ID (32-bit field that identifies the source of time for the data packet), the offset θ between its time and that of its reference ID, and more. Merging our zmap data with the openNTPproject *rv* data gave a total of 11,728,656 IPs that potentially run NTP servers.

OSes and clients in the wild. We use openNTPproject's *rv* data to get a sense of the

[8]NTP control query scans run in 2014 as part of (Czyz et al., 2014a)'s research found several 'mega-amplifiers': NTP servers that response to a single query with millions of responses. Our mode 3 scan also found a handful of these.

Table 2.2: Top ntpd versions in *rv* data from May 2015.

ntpd version	4.1.1	4.2.6	4.1.0	4.2.4	4.2.0	4.2.7	4.2.8	4.2.5	4.4.2
# servers	1,984,571	702,049	216,431	132,164	100,689	38,879	35,647	20,745	15,901

Table 2.3: Top OSes in *rv* data from May 2015.

OS	Unix	Cisco	Linux	BSD	Junos	Sun	Darwin	Vmkernal	Windows
# servers	1,820,957	1,602,993	835,779	38,188	12,779	6,021	3625	1994	1929

OSes and ntpd clients that are present in the wild. Importantly, the *rv* data is incomplete; *rv* queries may be dropped by firewalls and other middleboxes, NTP clients can be configured to refuse these queries, and some *rv* responses omit information. (This is why we had only 4M IPs in the *rv* data, while 10M IPs responded to our mode 3 zmap scan.) Nevertheless, we get some sense of what systems are out there by looking at the set of *rv* responses from May 2015. In terms of operating systems, Table 2.3 shows many servers running Unix, Cisco or Linux. Table 2.4 indicates that Linux kernels are commonly v2 (rather the more recent v3); in fact, Linux v3.0.8 was only the 13^{th} most popular Linux kernel. Meanwhile, Table 2.2 shows that ntpd v4.1.1 (released 2001) and v4.2.6 (released 2008) are most popular; the current release v4.2.8 (released 2014) is ranked only 8^{th} amongst the systems we see. The bottom line is that there are plenty of legacy NTP systems in the wild. As such, our lab experiments and attacks study the behavior of *two* NTP reference implementations: ntpd v4.2.6p5 (the second most popular version in our dataset) and ntpd v4.2.8p2 (the latest release as of May 2015).

Bad timekeepers. Next, we used our mode 3 zmap data to determine how many *bad timekeepers*–servers that are unfit to provide time—are seen in the wild. To do this, we compute the offset θ (equation (6.2)) for each IP that responded to our mode 3 queries, taking T_1 from the Ethernet frame time of the mode 3 query, T_4 from the Ethernet frame time of the mode 4 query, and T_2 and T_3 from the mode 4 NTP payload. We found many bad timekeepers — 1.7M had $\theta \geq 10$ sec, 3.2M had

Table 2.4: Top Linux kernels in *rv* data from May 2015.

kernel	2.6.18	2.4.23	2.6.32	2.4.20	2.6.19	2.4.18	2.6.27	2.6.36	2.2.13
# servers	123,780	108,828	97,168	90,025	71,581	68,583	61,301	45,055	29550

Table 2.5: Stratum distribution in our dataset.

stratum	0,16	1	2	3	4	5	6	7-10	11-15
# servers	3,176,142	115,357	1,947,776	5,354,922	1,277,942	615,633	162,162	218,370	187,348

stratum 0 or 16, and the union of both gives us a total of 3.7M bad timekeepers. Under normal conditions, NTP is great at discarding information from bad timekeepers, so it's unlikely that most of these servers are harming anyone other than themselves; we look into this in Sections 2.5.4-2.5.6.

Topology. Since a system's reference ID reveals the server from which it takes time, our scans allowed us to start building a subset of the NTP's hierarchical client-server topology. However, a reference ID only provides information about *one* of a client's preconfigured servers. In an effort to learn more, on June 28-30, 2015 we used nmap to send an additional mode 3 NTP query to every IP that had only one parent server in our topology; merging this with our existing data gave us a total of 13,076,290 IPs that potentially run NTP servers. We also wanted to learn more about the clients that synchronize to bad timekeepers. Thus, on July 1, 2015, we used the openNTPproject's scanning infrastructure to send a monlist query to each of the 1.7M servers with $\theta > 10$ sec. While monlist responses are now deactivated by many servers, because they have been used in DDoS amplification attacks (Czyz et al., 2014a), we

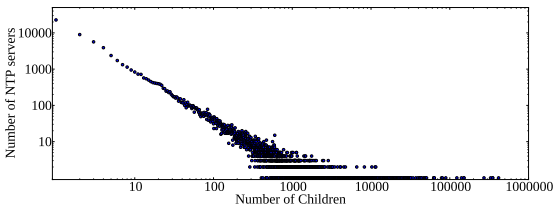

Figure 2·2: Client-degree distribution of NTP servers in our dataset; we omit servers with no clients.

did obtain responses from 22,230 of these bad timekeepers. Monlist responses are a trove of information, listing all IPs that had recently sent NTP packets (of any mode) to the server. Extracting only the mode 3 and 4 data from each monlist response, and combining it with our existing data, gave us a total of 13,099,361 potential NTP servers.

Stratum. Table 2.5 shows the distribution of stratums in our entire dataset. Note that there is *not* a one-to-one mapping between an NTP client and its stratum; because an NTP client can be configured with servers of various stratum, the client's stratum can change depending on the server it selects for synchronization. Thus, Table 2.5 presents the 'best' (*i.e.,* lowest) stratum for each IP in our dataset. Unsurprisingly, stratum 3 is most common, and, like (Czyz et al., 2014a) we find many unsynchronized (stratum 0 or 16) servers.

Degree distribution. Figure 2·2 shows the client (*i.e.,* child) degree distribution of the servers in our topology. We note that our topology is highly incomplete; it excludes information about clients behind a NAT or firewall, as well as servers that a client is configured for but not synchronized to.[9] The degree distribution is highly skewed. Of 13.1M IPs in our dataset, about 3.7M (27.8%) had clients below them in the NTP hierarchy. Of these 3.7M servers with clients, 99.4% of them have fewer than 10 clients, while only 0.2% of them have more than 100 clients. However, servers with more than 100 clients tend to have many clients, averaging above 1.5K clients per server, with the top 50 servers having at least 24.5K clients each. Compromising these important servers (or hijacking their traffic) can impact large swaths of the NTP ecosystem.

[9]Earlier studies (Murta et al., 2006), (Minar, 1999) used monlist responses, which are now commonly deactivated, to obtain topologies. We present a new technique that exposes a client's servers in Section 2.5.3, but as it is also a denial-of-service attack on the client, we have not used it to augment our measurements.

2.4 How to step time with NTP

Unauthenticated NTP traffic is vulnerable to on-path attacks, as was pointed out by various authors (Mizrahi, 2012b), (Haberman and Mills, 2010), (Klein, 2013), (Selvi, 2014), (Selvi, 2015). While on-path attacks are sometimes dismissed because the attacker requires a privileged position on the network, it is important to remember that an attacker can use various hijacking techniques to place herself on the path to an NTP server. For instance, ntpd configuration files allow clients to name servers by either their IP or their hostname (*e.g.,* MAC OS X 10.9.5 comes with an NTP client that is preconfigured to take time from the host `time.apple.com`, while many systems rely on the pool of servers that share the hostname `pool.ntp.org`). By hijacking the DNS entries for these hostnames (Kaminsky, 2008), (Herzberg and Shulman, 2013), an attacker can quietly manipulate the NTP traffic they send. Moreover, NTP relies on the correctness of IP addresses; thus ARP-spoofing in a local-area network or even global attacks on BGP (Goldberg, 2014) (like those seen in the wild (d'Itri, 2015), (Peterson, 2013)) can divert NTP traffic to an attacker.

In Section 2.2 and Table 2.1 we saw that dramatic shifts in time (years, months) are required when NTP attacks are used inside larger, more nefarious attacks. Can an on-path attacker really cause NTP clients to accept such dramatic shifts in time?

2.4.1 Time skimming

At first glance, the answer should be no. NTP defines a value called the *panic threshold* which is 1000 sec (about 16 minutes) by default; if NTP attempts to tell the client to alter its local clock by a value that exceeds the panic threshold, then the NTP client "SHOULD exit with a diagnostic message to the system log" (Mills et al., 2010). Our experiments confirm that ntpd v4.2.6 and v4.2.8 quit when they are initially synchronized to a server that then starts to offer time that exceeds the

panic threshold.

One way to circumvent this is through an adaption of (Selvi, 2014)'s "time-skimming" technique,[10] so that the man-in-the-middle slowly steps the client's local clock back/forward in steps smaller than the panic threshold. However, this comes with a caveat: it can take minutes or hours for ntpd to update a client's local clock. To understand why, observe that in addition to the panic threshold, NTP also defines a *step threshold* of 125 ms (Mills et al., 2010). A client will accept a time step larger than step threshold but smaller than the panic threshold as long as at least *"stepout"* seconds have elapsed since its last clock update; the *stepout* value is 900 seconds (15 minutes) in ntpd v4.2.6 and RFC 5905 (Mills et al., 2010), and was reduced to 300 seconds (5 minutes) in ntpd v4.2.8. Thus, shifting by one year using steps of size 16 minute each requires $\frac{1\times365\times24\times60}{16} = 33\text{K}$ total steps; with a 5-minute stepout, this takes at least 114 days.

2.4.2 Exploiting reboot.

There are other ways to quickly shift a client's time. ntpd has a configuration option called `-g`, which allows an NTP client that first initializes (*i.e.,* before it has synchronized to any time source) to accept *any time shift*, even one exceeding the panic threshold. (We have confirmed that both ntpd v4.2.6p5 and ntpd v4.2.8p2 on Ubuntu13.16.0-24-generic accept a single step 10 years back in time, and forward in time, upon reboot.) The `-g` configuration is quite natural for clocks that drift significantly when systems are powered down, and many OSes, including Linux, do run

[10]Selvi's (Selvi, 2014) time-skimming attack allows for fast timesteps on clients that update their clocks at predictable intervals—for instance, Fedora Linux sends a query every minute and updates its clock immediately upon receipt of the response. The full ntpd implementation has more complex clock update mechanisms that thwart this attack. These mechanisms include (1) sending mode 3 queries at randomized and adaptively-selected intervals determined by the poll process (Mills et al., 2010, Sec 13), (2) only updating the clock upon receipt of an adaptively-chosen number of self-consistent mode 4 responses (see *e.g.,* the discussion of TEST11 in Section 2.5.5), (3) using the stepout value to delay clock update events, *etc.*.

ntpd with `-g` by default.

Reboot. An on-path attacker can exploit `-g` by waiting until the client restarts as a result of power cycling, software updates, or other 'natural events'. Importantly, the attacker knows exactly when the client restarts, because the client puts 'INIT' in the reference ID of its packets (Figure 2·1), including the mode 3 queries the client sends the server. Moreover, a determined attacker can deliberately cause ntpd to restart using *e.g.,* a packet-of-death like Teardrop (BUGTRAQ mailing list, 1997).

Feel free to panic. Suppose, on the other hand, that an NTP attacker shifts a client's time beyond the panic threshold, causing the client to quit. If the operating system is configured to reboot the NTP client, the rebooted NTP client will initialize and accept whatever (bogus) time it obtains from its NTP servers. Indeed, this seems to have happened with some OSes during the November 2012 NTP incident (Menscher, 2012).[11]

Small-step-big-step. Users might notice strange shifts in time if they occur immediately upon reboot. However, we have found that ntpd allows an on-path attacker to shift time when clients are less likely to notice. To understand how, we need to look into ntpd's implementation of the `-g` configuration.

Small-step-big-step with ntpd v4.2.6. One might expect `-g` to allow for timesteps that exceed the panic threshold only upon initialization—when the client updates its clock for the very first time upon starting. To implement this, ntpd allows steps exceeding the panic threshold only when a variable called `allow_panic` is TRUE. ntpd v4.2.6p5 sets `allow_panic` to TRUE only upon initialization with the `-g` configuration (otherwise, it is initialized to FALSE), and set to FALSE if the client (1) is just about to update its local clock by a value less than the step threshold (125ms), and (2) is already in a state called SYNC, which means it recently updated its clock by a value less than the step threshold. Normally, a client initializes and (1) and (2) occur after

[11] Panic might be recorded in system log, but users may ignore it.

two clock updates. However, if an attacker is able to prevent the client from making ever two contiguous clock updates (one after the other) of less than 125 ms each, then `allow_panic` remains TRUE.

The following *small-step-big-step attack* on ntpd v4.2.6 exploits the above observation. First, the client reboots and begins initializing; it signals this to the server by putting 'INIT' in the reference ID of its mode 3 queries (Mills et al., 2010, Fig. 13)). Next, the client synchronizes to the server; it signals this with the server's IP address in the reference ID of its mode 3 queries (Mills et al., 2010, Fig. 13)). When the server sees that the client has synchronized once, the server sends the client a 'small step' greater than the STEP threshold (125 ms) and less than the panic threshold ($\approx$ 16 min); the client signals that it has accepted this timestep by putting 'STEP' in its reference ID (Mills et al., 2010, Fig. 13)). When the server sees that the client is in 'STEP' mode, the server immediately sends the client a 'big step' that exceeds the panic threshold. At this point, the client does not panic, because it never set `allow_panic` to FALSE. Indeed, one might even interpret this as expected behavior per RFC 5905 (Mills et al., 2010):

> STEP means the offset is less than the panic threshold, but greater than the step threshold STEPT (125 ms). In this case, the clock is stepped to the correct offset, but ... all associations MUST be reset and the client begins as at initial start.

Notice that this gives the server some 'slack time' before it sends the client the bogus big time step. We confirmed this for ntpd v4.2.6p5 with a small step of 10 minutes and a big step of 1 year. Upon initialization, our client exchanges 3 packets with the server before it first synchronizes, then 21 packets before accepting the small step and entering STEP mode, and 25 packets before accepting the big step.

Small-step-big-step with ntpd v4.2.8. Meanwhile, ntpd v4.2.8p2 sets `allow_panic`

to FALSE under conditions (1) and (2), OR if (1) holds and (3) the client is in FSET state, which is the state the client enters upon initialization. Normally, a client initializes and (1) and (3) occur after *one* clock update. Thus, small-step-big-step succeeds iff *every clock update* the client receives exceeds the step threshold (125ms). We confirmed this for an ntpd v4.2.8p2 client with a small step of 10 minutes and two big steps of 1 year. The very first mode 4 response received by the client, upon initialization, was the small step of 10 minutes back in time, and the client immediately went into 'STEP' mode. The next mode 4 response was a big step of 1 year back in time, which the client accepted after sending 11 queries to the server. The next mode 4 response was a another big step of 1 year, which the client accepted after sending 10 queries to the server.

Stealthy time shift. As an application of the small-step-big-step attack, an on-path attacker can preform a bogus big step and then to quickly bring the client's clock back to normal, so that the client never notices; this might be useful for stealthily flushing a client's cache, or expiring certain cryptographic objects (see Section 2.2). To do this, the attacker waits for ntpd to reboot (or deliberately causes a reboot), and ensures that every clock update made by the client makes is larger than 125 ms, sending the client into STEP mode. To keep things stealthy, the attacker can *e.g.,* first shift the client forward by 150 ms, then back by 150 ms, then forward by 150 ms, *etc.*. Then, when the attacker is ready, it can send a big step that exceeds the panic threshold, perform nefarious deeds, and finally send another big step that sets the client's clock back to normal.

2.4.3 Recommendation: Careful with `-g`

The security of ntpd should not rely on 100% OS uptime, so users should be careful with the `-g` option. One solution is to not use the `-g` option. Alternatively, one can detect feel-free-to-panic attacks by monitoring the system log for panic events and

being careful when automatically restarting ntpd after it quits. Monitoring should also be used to detect suspicious reboots of the OS (that might indicate the presence of a small-step-big-step or other reboot-based on-path attacks). Implementors can prevent small-step-big-step attacks by patching ntpd to ensure that the `allow_panic` variable is set to FALSE after the very first clock update upon initialization; this issue has been captured in CVE-2015-5300. Moreover, implementors can prevent ntpd clients from putting 'INIT' in the reference ID of their NTP packets upon initializing; this would make it more difficult for on-path attackers to know when initialization is taking place, raising the bar for attacks that exploit reboot.

2.5 Kiss-o'-Death: Off-path Denial-of-Service Attacks

We show how NTP security can be stymied by another aspect of the NTP protocol: the *'Kiss-o-death' (KoD)* packet. KoD packets are designed to reduce load at an NTP server by rate-limiting clients that query the server too frequently; upon receipt of a KoD from its server, the client refrains from querying that server for some period of time (Mills et al., 2010, Sec 7.4). We now show how KoD packets can be used to launch off-path denial-of-service attacks on NTP clients. We also consider using KoDs to pin a client to a bad timekeeper.

2.5.1 Why are off-path attacks hard?

We first need to understand why it is usually difficult to spoof NTP mode 4 packets (Figure 2·1) from off-path.

TEST2: The origin timestamp. Like many other protocols, NTP requires clients to check that a nonce in the client's query matches a nonce in the server's response; that way, an off-path attacker, that cannot observe client-server communications, does not know the nonce and thus has difficulty spoofing the packet. (This is analogous to source port randomization in TCP/UDP, sequence number randomization in TCP,

and transaction ID randomization in DNS.) NTP uses the *origin timestamp* as a nonce: the client checks that (a) the origin timestamp on the mode 4 response sent from server to client (Figure 2·1), matches (b) the client's local time when he sent the corresponding mode 3 query, which is sent in the *transmit timestamp* field of the mode 3 query sent from client to server. This is called `TEST2` in the ntpd code. (Note that ntpd does not randomize the UDP source port to create an additional nonce; instead, all NTP packets have UDP source port 123.)

How much entropy is in NTP's nonce? The origin timestamp is a 64 bit value, where the first 32 bits represent seconds elapsed since January 1, 1900, and the last 32 bits represent fractional seconds. A client whose system clock has *e.g.,* $\rho = -12$ bit precision ($2^{-12} = 244\mu$s) puts a $32 - 12 = 20$-bit random value in the least significant bit of the timestamp. Thus, for precision ρ, the origin timestamp has at least $32 + \rho$ bits of entropy. However, because polling intervals are no shorter than 16 seconds (Mills et al., 2010), an off-path attacker is unlikely to know exactly when the client sent its mode 3 query. We therefore suppose that the origin timestamp has about 32 bits of entropy. This is a lot of entropy, so one might conclude that NTP is robust to off-path attacks. However, in this section and Section 2.6, we will show that this is not the case.

2.5.2 Exploiting the Kiss-O'-Death Packet.

A server sends a client a Kiss-O'-Death (KoD) if a client queries it too many times within a specified time interval; the parameters for sending KoD are server dependent. The KoD is an NTP packet (Figure 2·1) with mode 4, leap indicator 3, stratum 0 and an ASCII 'kiss code' string in the reference ID field. A sample KoD packet is shown in Figure 2·3. According to RFC5905 (Mills et al., 2010):

> For kiss code RATE, the client MUST immediately reduce its polling

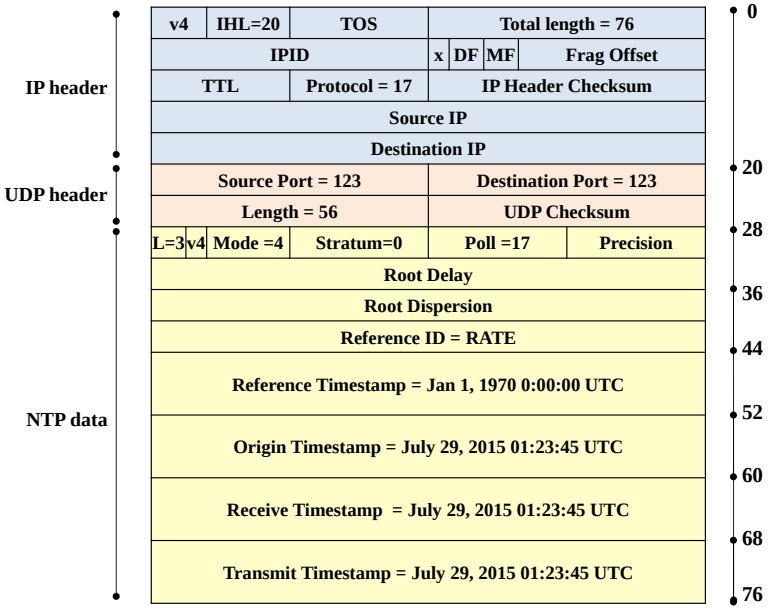

Figure 2·3: Kiss-o'-Death (KoD) packet, telling the client to keep quiet for at least 2^{17} seconds (36 hours).

interval to that server and continue to reduce it each time it receives a RATE kiss code.

In ntpd v4.2.6 and v4.2.8, this is implemented by having the client stop querying the server for a period that is at least as long as the poll value field in the received KoD packet.[12] Our experiments confirm that if the KoD packet has polling interval $\tau_{\text{kod}} = 17$ then the ntpd v4.2.8 client will stop querying the server for at least $2^{\tau_{\text{kod}}}$ sec (36 hours).[13] The poll field in the NTP packet is an 8-bit value (*i.e.,* ranging from 0 to 255), but RFC 5905 (Mills et al., 2010, pg 10) defines the maximum allowable poll

[12]Interestingly, RFC 5905 (Mills et al., 2010, Sec. 7.4) defines an even more dangerous type of KoD packet: " For kiss codes DENY and RSTR, the client MUST demobilize any associations to that server and stop sending packets to that server". Thus, spoofing a single DENY or RSTR KoD packet can completely disconnect a client from its server! Fortunately, however, neither ntpd v4.2.6 or v4.2.8 honor this functionality.

[13]Due to complex interactions between τ_{kod}, the poll value in the KoD packet, and NTP's polling algorithm, the period of time that the client stops querying the server will usually exceed $2^{\tau_{\text{kod}}}$. More precisely, the client's polling algorithm resets the minpoll value τ_{min} for the server sending the KoD to $\tau_{\text{min}} = \max(\tau_{\text{min}}, \tau_{\text{kod}})$ and then the client stops querying the server for a period of about $2^{\max(\tau_{\text{min}}+1, \max(\min(\tau_{\text{max}}, \tau_{\text{kod}})))}$. By default, minpoll is initialized to $\tau_{\text{min}} = 6$, and maxpoll to $\tau_{\text{max}} = 10$.

value to be 17. The most recent ntpd implementation, however, will accept KoDs with poll values even larger than 17; setting $\tau_{\text{kod}} = 25$, for example, should cause the client to stop querying its server for at least 2^{25} seconds, *i.e.,* ≈ 1 year.

Spoofing a KoD from off-path. How does the client know that the KoD packet came from the legitimate server, and not from an attacker? With regular mode 4 responses, the client uses the origin timestamp as a nonce. While it seems reasonable to expect this check to be performed on the KoD packet as well, RFC 5905 (Mills et al., 2010, Sec. 7.4) does not seem to explicitly require this. Moreover, lab experiments with ntpd v4.2.6p5 and v4.2.8p3 show that the client accepts a KoD even if its origin timestamp is bogus. This means that an offpath attacker can trivially send the client a KoD that is spoofed to look like it came from its server; the only information the attacker needs is the IP addresses of the relevant client and server. Moreover, by setting the poll value in the spoofed KoD to be an unreasonably high value (*e.g.,* $\tau_{\text{kod}} = 25$), the spoofed KoD will prevent the client from querying its server for an extended period of time. This vulnerability, captured in CVE-2015-7704, was patched in ntpd v4.2.8p4 after the disclosure of our work.

Eliciting a KoD from off-path: Priming the pump. Even if the client does validate the origin timestamp on the KoD packet, an off-path attacker could still *elicit* a valid KoD packet for the client from the server. To do this, the off-path attacker 'primes-the-pump' at the server, by sending multiple mode 3 queries spoofed to look like they come from the victim client; the server then 'gets angry' at the client, and responds to the client's legitimate mode 3 queries with a KoD. The client will accept this KoD because it has a valid origin timestamp (matching that in the client's legitimate query). The attacker just needs to measure the number of times q in a given period of time t_0 that a client must query the server in order to elicit a KoD; this is easily done by attempting to get the server to send KoD packets to the attacker's own machine.

Then, the attacker sends the server $q-1$ queries with source IP spoofed to that of the victim client, and hopes that the client sends its own query to server before time t_0 elapses. This issue has been captured in CVE-2015-7705. Interestingly, recent NTP security bulletins (as of 09/2015) have increased this attack surface by recommending that servers send KoDs (Axel K, 2015).[14]

Attack efficiency. A *single* spoofed KoD packet with high poll (*e.g.,* $\tau_{\text{kod}} \geq 25$) can essentially prevent the client from ever taking time from its server, and thus gives rise to a very low-rate off-path denial-of-service attack (CVE-2015-7704). Meanwhile, eliciting a KoD by priming-the-pump (CVE-2015-7705) requires more packets, because:

1) The attacker must send several packets to the server to elicit the KoD, and

2) The elicited KoD packet is likely to have poll value no larger than 10. (This follows because when ntpd servers send KoDs, the KoD's poll value τ_{kod} is at least as large as that in the query triggering the KoD. Meanwhile, the query triggering the KoD was a legitimate query from the client, and default maximum value of the poll in the client's query is 10.) The attacker can thus elicit a KoD that quiets the client for $\approx 2^{10}$ seconds (15 minutes), then elicit another KoD 15 minutes later, and continue this indefinitely.

Thus, patching clients to validate the KoD origin timestamp (CVE-2015-7704) does weaken, but does not eliminate, KoD-related attacks.

2.5.3 Low-rate off-path denial-of-service attack on NTP clients.

It's tempting to argue that NTP clients are commonly pre-configured with many, and so KoD-related attacks on one server can be mitigated by the presence of the other servers. However, this is not the case. We now present a denial-of-service attack that

[14] As of August 2015, (Axel K, 2015) recommends the configuration `restrict default limited kod nomodify notrap nopeer`. Note that turning on `limited` means that a server will not serve time to a client that queries it too frequently; `kod` additionally configures the server to send KoDs.

allows an off-path attacker to "turn off" NTP at a client by preventing the client from synchronizing to any of its preconfigured servers.

What are the implications of this attack? For the most part, the client will just sit there and rely on its own local clock for the time. If the client has accurate local time-keeping abilities, then this attack is unlikely to cause much damage. On the other hand, the client machine could be incapable of keeping time for itself, *e.g.,* because it is in a virtual machine (VMware, 2011), or running CPU-intensive operations that induce clock drift. In this case, the client's clock will drift along, uncorrected by NTP, for the duration of attack.

The denial of service attack. The attack proceeds as follows:

1) The attacker sends a mode 3 NTP query to the victim client, and the client replies with a mode 4 NTP response. The attacker uses the reference ID in the mode 4 response to learn the IP of the server to which the client is synchronized.

2) The attacker spoofs/elicits a KoD packet with τ_{kod} from the server to which the client is synchronized. The client stops querying this server for at least $2^{\tau_{\text{kod}}}$ sec.

3) There are now two cases. Case 1: the client declines to take time from any its of other preconfigured servers; thus, the attacker has succeeded in deactivating NTP at the client. Case 2: The client will synchronize to one of its other preconfigured servers, and the attacker returns to step 1. To determine whether the client is in the Case 1 or Case 2, the attacker periodically sends mode 3 queries to the client, and checks if the reference ID in the mode 4 response has changed.

Thus, the attacker learns the IP addresses of all the preconfigured servers from which the client is willing to take time, and periodically (*e.g.,* once every $2^{\tau_{\text{kod}}}$ seconds), spoofs/elicits KoD packets from each of them. The client will not synchronize to any of its servers, and NTP is deactivated.

Attack surface. For this attack to work, the client must (1) react to KoD packets by

refraining from querying the KoD-sending server, (2) respond to NTP mode 3 queries with NTP mode 4 responses, and (3) be synchronized to an IPv4 NTP server. This creates a large attack surface: condition (1) holds for ntpd v4.2.6 and v4.2.8p3, the most recent reference implementation of NTP, and our scans (Section 2.3.2) suggest that over 13M IPs satisfy condition (2).

Sample experiment. We ran this attack on an ntpd v4.2.8p2 client in our lab configured with the IP addresses of three NTP servers in the wild. We elicited a KoD for each server in turn, waiting for the client that resynchronize to a new server before eliciting a new KoD from that server. To elicit a KoD, a (separate) attack machine in our lab ran a scapy script that sent the server 90 mode 3 queries in rapid succession, each of which was spoofed with the source IP of our victim client and origin timestamp equal to the current time on the attacker's machine. (Notice that the origin timestamp is bogus from the perspective of the victim client.) Because our spoofed mode 3 queries had poll value $\tau = 17$, the elicited KoDs had poll value $\tau = 17$. Our client received its third KoD within 1.5 hours, and stopped querying its servers for the requested 2^{17} seconds (36 hours); in fact, the client had been quiet for 50 hours when we stopped the experiment.

2.5.4 Pinning to a bad timekeeper. (Part 1: The attack)

Consider a client that is preconfigured with several servers, one of which is a bad timekeeper. Can KoDs force the client to synchronize to the bad timekeeper? Indeed, since ntpd clients and configurations are rarely updated (see Table 2.2, that shows that 1.9M servers use a version of ntpd that was released in 2001), a bad timekeeper might be lurking in rarely-updated lists of preconfigured servers.

The attack. The attacker uses the KoD denial-of-service attack (Section 2.5.3) to learn the client's preconfigured servers; each time the attacker learns a new server, the attacker sends a mode 3 query to the server to check if it is a good timekeeper,

continuing to spoof KoDs until it identifies a server that is a bad timekeeper. At this point, the client is taking time from a bad timekeeper, and the attack succeeds. To ensure that the client remains pinned to the bad timekeeper, the attacker periodically sends the client fresh KoDs from the good timekeepers.

But does this attack actually work? NTP's clock discipline algorithms are designed to guard against attacks of this type. We have launched this attack against different clients (in our lab) configured to servers (in the wild), and observed differing results; sometimes, that client does take time from the bad timekeeper, and sometimes it does not. Unfortunately, however, we have not yet understood exactly what conditions are required for this attack to succeed. One thing we do know, however, are conditions that would definitely cause this attack to fail. We explain these conditions by taking detour into some aspects of NTP's clock discipline algorithm, that will also be important for the attacks in Section 2.6.

2.5.5 Detour: NTP's clock discipline algorithms.

An NTP client only updates its local clock from its chosen server at infrequent intervals. Each valid mode 4 response that the client obtains from its server is a *sample* of the timing information at that server. A client needs to obtain enough "good" samples from a server before it even considers taking time from that server.

There are mechanisms that "age", but do not necessarily delete, old samples from servers that have not been heard from in a long time (because *e.g.,* they have an excessively long polling interval, or have been KoD'd). Empty samples are recorded under various failure conditions; we discussed one such failure condition, `TEST2` (origin timestamp validity) in Section 2.5.1. `TEST7` is another important failure condition:

TEST7. Read the root delay Δ and root dispersion E from the server's mode 4 packet (Figure 2·1) and check that $\Delta/2 + E$ does not exceed MAXDISP, a parameter whose default is 16 sec (Mills et al., 2010). (Roughly, $\Delta/2 + E$ measures how far off

the server's clock is, relative to the stratum 1 server that is at the 'root' of the NTP hierarchy from which the server is taking time.)

For each non-empty sample, the client records the offset θ per equation (6.2) and delay δ per equation (6.1). The client keeps up to eight samples from each server, and selects the offset θ^* corresponding to the non-empty sample of lowest delay δ^*. It then computes the *jitter* ψ, which is the root-mean-square distance of the sample offsets from θ^*, *i.e.,*

$$\psi = \sqrt{\tfrac{1}{i-1}\sum_i (\theta_i - \theta^*)^2} \tag{2.3}$$

Next, the server must pass another crucial check:

TEST11. Check that the *root distance* Λ does not exceed MAXDIST, a parameter that defaults to 1.5 sec. While RFC 5905 (Mills et al., 2010, Appendix A.5.5.2), ntpd v4.2.6 and ntpd v4.2.8 each use a slightly different definition of Λ, what matters is

$$\Lambda \;\propto\; \psi + (\delta^* + \Delta)/2 + E + 2^{\rho} \tag{2.4}$$

where Δ is the *root delay*, E is the *root dispersion*, and ρ is the *precision*, all which are read off the server's mode 4 packet per Figure 2·1. (Precision ρ reveals the resolution of the server's local clock; *e.g.,* $\rho = -12$ means the server's local clock is precise to within 2^{-12} sec or 244 μs. Root delay Δ is the cumulative delay from the server to the 'root' (*i.e.,* stratum 1 server) in the NTP client-server hierarchy from which the server is taking time; a stratum 1 server typically has $\Delta = 0$. Root dispersion E is an implementation-dependent quantity related to the Λ value computed by the server.)

If the client has several servers that pass `TEST11` (and other tests we omit here), the client must select a single server to synchronize its clock. This is done with a variant of Marzullo's Algorithm (Marzullo, 1984), which clusters servers according to offset θ^* (equation (6.2)) and other metrics. Servers who differ too much from the majority are ignored, while the remaining servers are added to a 'survivor list'. (This is why,

under normal conditions, NTP clients do not synchronize to bad timekeepers.) If all servers on the survivor list are *different* from the server the client used for its most recent clock update, the client must decide whether or not to "*clock hop*" to a new server. The clock hop decision is made by various other algorithms that are different in ntpd v4.2.6 and v4.2.8. If no clock hop occurs, the client does not update its clock.

2.5.6 Pinning to a bad timekeeper. (Part 2: Attack surface)

Thus, for a client to synchronize to a bad timekeeper, we know that the bad timekeeper must be able to pass `TEST11`. In practice, we found that this means that bad timekeeper must send mode 4 packets (Figure 2·1) with root delay Δ and root dispersion E and precision ρ such that $\Delta/2 + E + 2^{\rho} < 1$ sec. In addition to this, the bad timekeeper must (1) "defeat" the good timekeepers in Marzullo's algorithm, and then (2) convince the client to clock hop.

Example: A successful attack. We synchronized an ntpd v4.2.6 client in our lab to three stratum 2 servers in the wild. Two of these servers were good timekeepers, and the bad timekeeper had an offset of -224 sec relative to our client with $E \approx 10$ msec and $\Delta \approx 0$. We let the client synchronize to one of the good timekeepers, and then an (separate) attack machine in our lab elicited KoDs with poll interval $\tau = 17$ (36 hours) from the two good timekeepers (by sending 90 NTP mode 3 packets spoofed with the source IP of our client in rapid succession to each). At this point, the client stopped querying the good timekeepers, still keeping the old samples from these servers in its logs, and continuously getting fresh samples from the bad timekeeper. At this time, the client indicated that Marzullo's Algorithm was eliminating the bad timekeeper from consideration. Then, 94 minutes after the KoDs were first sent, the client cleared its logs of samples and sent one query to each of its three servers. Because the good timekeepers had poll interval of $\tau = 17$, the client stopped querying them after this one query. Meanwhile, the client continued to get fresh samples from

the bad timekeeper. Once the client had four fresh samples from the bad timekeeper, it was accepted for synchronization.

This experiment is quite curious; we are not sure what caused the client to clear its log of samples after 94 minutes and requery all its servers. Once this happened, the larger number of fresh samples logged from the bad timekeeper allowed it to become the chosen server for synchronization. Indeed, we ran the exact same experiment (using the same servers) with a ntpd v4.2.8 client, and after six hours the client never cleared its log of samples, and thus never accepted time from the bad timekeeper. We tried the same experiment with the same ntpd v4.2.6 client and good timekeepers, but this time using a 'worse' timekeeper with offset -582s and $E \approx 10$ msec and $\Delta \approx 0$; five hours later, the client still had not cleared its logs or taken time from the bad timekeeper.

KoD + reboot. There is one way around all of the issues discussed so far. If ntpd reboots,[15] then the attacker has a short window during which the client is not synchronized to *any* of its preconfigured servers; our experiments confirm that in ntpd v4.2.6 this window is at least 4 polling intervals (*i.e.,* the minimum polling interval is 2^4 sec, so this translates to at least 1 min), while ntpd v4.2.8 shortens this to after the client receives its first mode 4 responses from its servers. Thus, the attacker can exploit this short window of time to spoof a KoD packet from each of the clients' good-timekeeper servers, but not from the bad timekeeper. As long as the bad timekeeper passes `TEST11`, the attacker no longer has to worry about Marzullo's algorithm, clock-hopping, or exceeding the panic threshold, and the attack will succeed.

Bad timekeepers in the wild. To quantify the attack surface resulting from pinning to a bad timekeeper, we checked which bad timekeepers (with offset $\theta > 10$ sec) are

[15] A reboot can be elicited by sending a packet-of-death that restarts the OS (*e.g.,* Teardrop (BUGTRAQ mailing list, 1997)) or the ntpd client (*e.g.,* CVE-2014-9295 (National Vulnerability Database, 2014)). Alternatively, the attacker can wait until the client reboots due to a power cycling, software updates, or other 'natural' events.

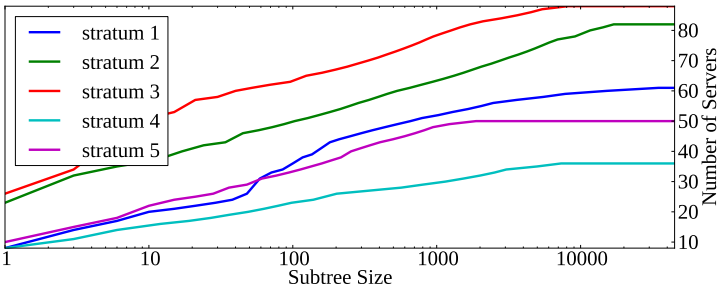

Figure 2·4: Cumulative distribution of the size of the subtrees rooted at bad timekeepers (with offset $\theta > 10$ sec) that can pass `TEST11` (because $\Delta/2 + E + 2^{\rho} < 1$ sec), broken out by the bad timekeeper's stratum. We omit bad timekeepers with no children.

"good enough" to have (a) $\Delta/2 + E + 2^{\rho} < 1$ sec and (b) stratum that is not 0 or 16. Of the 3.7M bad timekeepers in our dataset (Section 2.3.2), about 368K (or 10%) are "good enough" to pass the necessary tests. Meanwhile, we found only 2190 bad timekeepers that had clients below them in the NTP hierarchy, and of these, 743 were "good enough". While our topology is necessarily incomplete (Section 2.3.2), this suggests that there are only a limited number of servers in the wild that can be used for these attacks. Figure 2·4 is a cumulative distribution of the size of the subtrees rooted at bad timekeepers that are "good enough" to pass the necessary tests, broken out by the bad timekeeper's stratum. (Point (x, y) in the figure corresponds to y "good-enough" bad timekeepers have a subtree of size at least x.) Notice, however, that the distributions are highly skewed, so some of the "good enough" bad timekeepers had large subtrees below them in the NTP hierarchy. For example, one stratum 1 server (in Central Europe) had an offset of 22 minutes and almost 20K clients in its subtree.

2.5.7 Recommendation: Kiss-o'-Death considered harmful.

Following the disclosure of our work, many NTP implementations (including ntpd v4.2.8p4) have been patched to ensure that clients validate the origin timestamp on KoD packets (CVE-2015-7704), thus preventing our extremely low-rate off-path

denial-of-service attack that uses spoofed KoD. However, our less severe priming-the-pump attack that allows the attacker to elicit a valid KoD (CVE-2015-7705) has not yet been addressed. Preventing this attack requires some rethinking of NTP's KoD and rate-limiting functionality.

One solution is to simply eliminate NTP's KoD and other rate-limiting functionality; this, however, eliminates a server's ability to deal with heavy volumes of NTP traffic.

Alternatively, if clients are required to cryptographically authenticate their queries to the server, then it is no longer possible for an off-path attacker to prime the pump at the server by spoofing mode 3 queries from the client. Interestingly, however, a new proposal for securing NTP (Sibold et al., 2015, Sec.4) only suggests authenticating mode 4 responses from the server to client, but not mode 3 queries from client to server.

In the absence of authentication, another solution is to apply techniques developed for rate-limiting other protocols, *e.g.,* Response Rate Limiting (RRL) in the DNS (Vixie, 2014). With RRL, nameservers do not respond to queries from clients that query them too frequently.[16] Like NTP, DNS is sent over unauthenticated UDP, and therefore is at risk for the same priming-the-pump attacks we discussed here. RRL addresses this by requiring a server to randomly respond to some fraction of the client's queries, even if that client is rate limited (Vixie and Schryver, 2012, Sec. 2.2.7); thus, even a client that is subject to a priming-the-pump attack can still get some good information from the server. To apply this to NTP, a server that is rate-limiting a client with KoDs would send legitimate mode 4 responses (instead of a KoD) to the client's queries with some probability. For this to be effective, NTP clients should also limit the period for which they are willing to keep quiet upon receipt of a KoD;

[16] ntpd also offers this type of rate limiting, as an alternative to the KoD, via the `limited` configuration option.

not querying the server for days ($\tau_{\text{kod}} = 17$) or even years ($\tau_{\text{kod}} = 25$) upon receipt of a single KoD packet is excessive and invites abuse.

2.6 Off-Path NTP Fragmentation Attack

We now show how an *off-path* attacker can hijack an unauthenticated NTP connection from a client to its server. The key ingredient in our attack is *overlapping IPv4 packet fragments*; therefore this attack succeeds a on small but non-negligible set of clients and servers that use the IPv4 fragmentation policies described in Section 2.6.5. We will assume the client is preconfigured with only one server. (Some OSes (*e.g.,* MAC OS X v10.9.5) actually do use this configuration, and in Section 2.5.6 we show how to combine our KoD and reboot techniques to simulate this scenario for clients preconfigured with multiple servers.) We first explain why off-path attacks are challenging and give background on IPv4 packet fragmentation (Postel, 1981a), (Novak, 2005), (Shankar and Paxson, 2003), (Ptacek and Newsham, 1998). Next, we present the attack itself, explain when it works, and conclude with a measurement study that sheds light on the number of clients and servers in the wild that are vulnerable to this attack.

2.6.1 Why are off-path attacks hard?

The goal of our attacker is to spoof a series of mode 4 response packets (Figure 2·1) from the server to the client. The spoofed response should contain bogus server timestamps (*i.e.,* T_3 transmit timestamp, T_2 receive timestamp) in order to convince the client to accept bogus time from the server. This creates several hurdles for an off-path attacker who cannot see the communication between client and server:

First, there is the issue of nonces. Per Section 2.5.1, the attacker must spoof packets with the correct origin timestamp, which has about 32 bits of entropy. Our off-path attacker will not even try to learn the origin timestamp; instead, we use the

origin timestamp from the honest mode 4 response from server to client, and use IPv4 packet fragmentation to overwrite other relevant fields of the NTP packet.

Second, since our attacker does not know NTP's origin timestamp, it cannot compute the UDP checksum. However, the UDP specification for IPv4 allows a host to accept any packet with UDP checksum of zero (which means: don't bother checking the checksum) (Postel, 1980, pg 2). As such, our attacker uses IPv4 fragmentation to set the UDP checksum to zero.

Third, in order to convince the client's clock discipline algorithms (Section 2.5.5) to accept the attacker's bogus time, our attacker must spoof a *stream* of several (at least eight, but usually more) packets that are acceptable to the clock discipline algorithm. This is significantly more challenging than just spoofing a *single* packet as in *e.g.,* (Kaminsky, 2008), (Herzberg and Shulman, 2013). Moreover, this stream of spoofed packets must be sufficiently self-consistent to pass `TEST11` (Section 2.5.5). To do this, our attacker uses IPv4 fragmentation to set several fields in the packet to tiny values (*e.g.,* $\rho = -29$, $\Delta = 0.002$, $E = 0.003$ sec and stratum $= 1$). His main challenge, however, is to ensure that jitter ψ is small enough to pass TEST11. We show how this can be done in Section 2.6.4.

2.6.2 IPv4 packet fragmentation.

Fragmentation is one of IP's original functionalities (Postel, 1981a); chopping a large packet into fragments that can be sent through networks that can only handle small packets. The length of the largest packet that a network element can accommodate is its '*maximum transmission unit (MTU)*'. In practice, almost every network element today supports an MTU of at least 1492 bytes (the maximum payload size for Ethernet v2 (Mamakos et al., 1999, Sec. 7)). Back in 1981, however, RFC791 (Postel, 1981a) required that "all hosts must be prepared to accept" IPv4 packets of length 576 bytes, while "every internet module must be able to forward" IPv4 packets of length

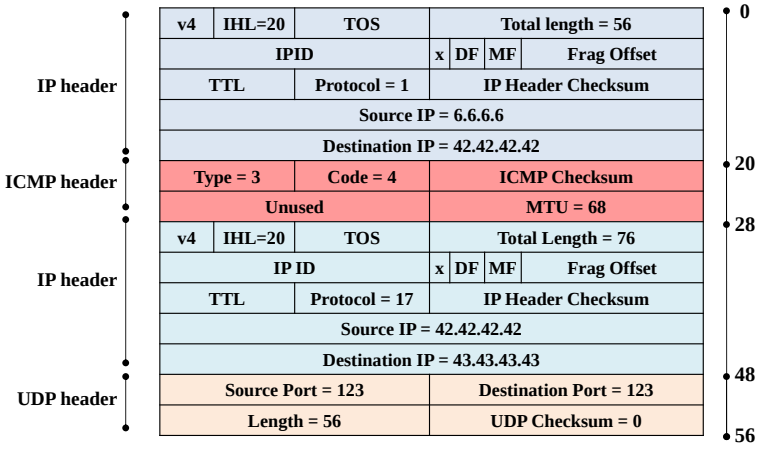

Figure 2·5: ICMP Fragmentation Needed packet from attacker 6.6.6.6 telling server 42.42.42.42 to fragment NTP packets for client 43.43.43.43 to MTU of 68 bytes.

68 bytes. The minimum IPv4 MTU for the Internet is therefore 68 bytes, but many OSes refuse to fragment packets to MTUs smaller than 576 bytes. Our attack only succeeds against servers that fragment to a 68 byte MTU; the attacker can therefore convince a server to chop an NTP packet into the two fragments of Figure 2·7. Our measurements (Section 2.6.7) confirm that there are ten of thousands of NTP servers in the wild that do this.

ICMP Fragmentation Needed. How does a host learn that it needs to fragment packets to a specific MTU? Any network element on the path from sender to receiver can send a single *ICMP fragmentation needed* packet to the sender containing the desired MTU; this information is then cached for some OS-dependent period of time (*e.g.,* 10 minutes by default on Linux 3.13.0 and MAC OS X 10.9.5). Figure 2·5 shows an *ICMP fragmentation needed* packet that signals to host 42.42.42.42 to fragment all NTP packets (UDP port 123) it sends to destination IP 43.43.43.43 to an MTU of 68 bytes. Since the host is not expected to know the IP addresses of all the network elements on its path, this packet can be sent from any source IP; in Figure 2·5 this source IP is 6.6.6.6. The payload of this ICMP packet contains an IP header and first eight bytes of a packet that has already been sent by host and exceeded the

MTU (Postel, 1981b); for NTP, these eight bytes correspond to the UDP header. The sender uses this to determine which destination IP (*i.e.,* 43.43.43.43) and protocol (*i.e.,* UDP port 123) requires fragmentation. Our attacker (at IP 6.6.6.6) can easily send an *ICMP fragmentation needed* from off-path. Its only challenge is (1) choosing UDP checksum (which it sets to zero) and (2) matching the IPID in the ICMP payload with that in an NTP packet previously sent to the client (which it can do, see Section 2.6.4, and moreover some OSes don't bother checking this (*e.g.,* Linux 3.13.0)).

IPv4 Fragmentation. How do we know that an IPv4 packet is a fragment? Three IPv4 fields are relevant (see Figure 2·1). *Fragment offset* specifies the offset of a particular fragment relative to the beginning of the original unfragmented IP packet; the first fragment has an offset of zero. The *more fragment (MF)* flag is set for every fragment except the last fragment. *IPID* indicates that a set of fragments all belong to the same original IP packet. Our attacker infers IPID (Section 2.6.4), and then sends the client spoofed IPv4 fragments with the same IPID as the legitimate fragments sent from the server, as in Figure 2·6. The spoofed and legitimate fragments are reassembled by the client into a single crafted NTP packet.

Fragment reassembly. How does a host reassemble a fragmented IPv4 packet? In the common case, the fragments are *non-overlapping*, so that the offset of one fragment begins immediately after the previous fragment ends. In this case, the host checks its *fragment buffer* for fragments with the same IPID, and pastes their payloads together according to their fragment offset, checking that the last fragment has a MF=0 (Postel, 1981a). Fragment buffer implementations differ in different OSes (Knockel and Crandall, 2014; Herzberg and Shulman, 2013). Meanwhile, the RFCs are mostly silent about reassembly of *overlapping* fragments, like the ones in Figure 2·6 and 2·7.[17] Several authors (Ptacek and Newsham, 1998), (Shankar

[17]RFC 3128 (Miller, 2001) does have some specific recommendations for overlapping IPv4 frag-

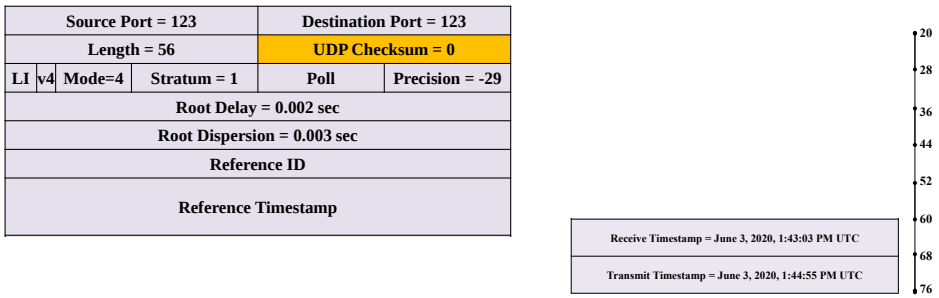

Figure 2·6: IPv4 fragments for our attack: 1^{st} and 2^{nd} spoofed fragments.

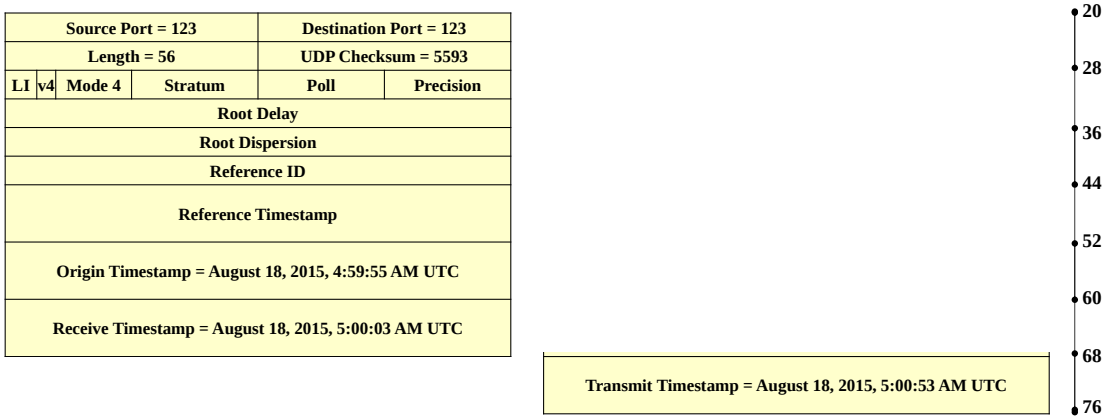

Figure 2·7: IPv4 fragments for our attack: 1^{st} and 2^{nd} legitimate fragments.

and Paxson, 2003), (Novak, 2005), (Baggett, 2012) have observed that reassembly policies differ for different operating systems, and have undertaken to determine these policies using clever network measurement tricks. (Hilariously, Wireshark has an overlapping fragment reassembly policy that is independent of its host OS (Baggett, 2012) and is therefore useless for this purpose.) Our attacks also rely on overlapping fragments. Overlapping fragment reassembly policies are surprisingly complex, poorly documented, and have changed over time. Thus, instead of generically describing them, we just consider reassembly for the specific fragments used in our attack.

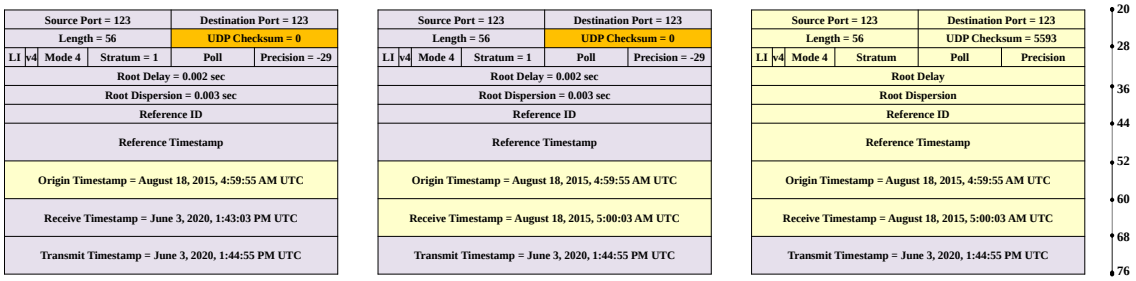

Figure 2·8: Different ways our fragments may be reassembled. From left to right: Outcome A, Outcome B, Outcome C.

2.6.3 Exploiting overlapping IPv4 fragments.

Our attack proceeds as follows. The attacker sends the server a spoofed *ICMP fragmentation needed* packet (Figure 2·5) requesting fragmentation to a 68-byte MTU for all NTP packets sent to the client. If the server is willing to fragment to a 68-byte MTU, the server sends all of its mode 4 NTP responses as the two fragments in Figure 2·7. Meanwhile, our attacker plants the two spoofed fragments in Figure 2·6 in the client's fragment buffer. The spoofed fragments sit in the fragment buffer and wait for the server's legitimate fragments to arrive. The first spoofed fragment sets the UDP checksum to zero, and sets $\rho = -29$, $\Delta = 0.002$, $E = 0.003$ to tiny values so that the reassembled packet is more likely to pass TEST11. The second spoofed fragment sets the NTP receive timestamp (T_2) and transmit timestamps (T_3) to bogus time values. Both spoofed fragments must have the same IPID as the two legitimate fragments; we explain how to do this in Section 2.6.4. This process of planting spoofed fragments continues for every mode 4 NTP response that the server sends the client. Once the client has accepted the bogus time, the attacker spoofs KoDs (Section 2.5.3) so the client stops updating its clock. (The attacker can check that the client accepted the bogus time by sending it a mode 3 query and checking the timestamps in the client's mode 4 response.)

ments in the case of TCP; NTP, however, is sent over UDP. Also, overlapping fragments are forbidden for IPv6.

The victim client receives the four overlapping fragments in Figure 2·6 and 2·7, in the order shown, with the first fragment in 2·6 arriving earliest. How are they reassembled? One potential outcome is for the client to reject the fragments altogether because they are overlapping or too short. Otherwise, the first honest fragment arrives in the client's fragment buffer and is reassembled with one or both of the spoofed fragments, according to one of the reassembly outcomes shown in Figure 2·8. (Note: the second honest fragment will not reassemble with anything, because by the time it arrives in the buffer the earlier fragments have already reassembled.) In Outcome A the client prefers fragments that arrive earliest, pasting the first legitimate fragment underneath the two spoofed fragments that were waiting in the cache (*i.e.,* the 'First' policy in the Paxson/Shankar overlapping-packet reassembly model (Shankar and Paxson, 2003), and the 'First', 'Windows' and 'Solaris' policies of Novak (Novak, 2005)). In Outcome B, the client prefers an earlier fragment with an offset that is less than or equal to a subsequent fragment (*i.e.,* the 'BSD' policy of (Shankar and Paxson, 2003), (Novak, 2005)). In Outcome C the client prefers fragments that arrive later over those that arrive earlier (*i.e.,* the 'Last' and 'Linux' policies of (Shankar and Paxson, 2003), (Novak, 2005)).

In which outcome does our attack succeed? In Outcome C, the packet is dropped due to its incorrect UDP checksum, and our attack fails. In Outcomes A and B, our off-path attacker successfully injects packets with the correct origin timestamp and UDP checksum. However, in Outcome B, the attacker controls only the transmit timestamp T_3 in the reassembled packet. Because passing `TEST11` (equation (2.4)) constrains delay δ to be < 1 sec (equation (6.1)) it follows that the spoofed T_3 must be within 1 sec of the legitimate receive timestamp T_2, making our attack much less interesting. Our attack therefore works best in Outcome A, where the attacker controls *both* the T_3 and T_2; by setting $T_2 \approx T_3$, the delay δ is small enough to pass

`TEST11`, even if the spoofed T_2 and T_3 are very far from the legitimate time.

2.6.4 Planting spoofed fragments in the fragment buffer.

Because a client will only take time from a server that provides several self-consistent time samples (Section 2.6.1), our attacker must craft a *stream* of NTP mode 4 responses. In achieving this, our attacker must surmount two key hurdles:

Hurdle 1: Jitter. The reassembled *stream* of packets must be sufficiently self-consistent to pass `TEST11`, so that root distance $\Lambda < 1.5$ sec (equation (2.4)). If the client reassembles packets as in Outcome A, the attacker can set tiny values for the precision ρ, root delay Δ and root dispersion E (in the first spoofed fragment on the left of Figure 2·6) and delay δ (by setting $T_2 \approx T_3$ in the second spoofed fragment). What remains is ensuring that jitter ψ is sufficiently small (equation (5.3)); this means that the offset values θ in the reassembled stream of packets must be consistent to within about 1 sec.

Why is this difficult? The key problem is that the offset θ is determined by the timestamps T_2 and T_3 set in the attacker's second spoofed fragment (Figure 2·6), as well as the origin and destination timestamps T_1, T_4. T_1 corresponds to the moment when the legitimate client sends its query, and is unknown to our off-path attacker. Moreover, T_1 roughly determines T_4, which roughly corresponds to the moment when the first legitimate fragment reassembles with the spoofed fragments in the client's fragment buffer. Now suppose the fragment buffer caches for 30 sec. This means that timestamps T_2 and T_3 (from attacker's second spoofed fragment) can sit in the fragment buffer for anywhere from 0 to 30 sec before reassembly at time T_4 (Figure 2·6). Thus, the offset θ in the reassembled packet can vary from 0 to 30 sec, so that jitter $\psi \approx 30$ sec and the attack fails `TEST11`.

Hurdle 2: IPID. Our attacker must ensure that the IPID of the spoofed fragments planted in the fragment buffer (the two fragments in Figure 2·6) match the IPID of

the legitimate fragments sent by the server (the two fragments in Figure 2·7); this way, the spoofed fragments will properly reassemble with the legitimate fragments.

Surmounting these hurdles. To surmount the first hurdle, our attacker ensures that the client's fragment buffer always contains fresh copies of the second spoofed fragment that are no more than 1 sec old. Suppose that the client's fragment cache is a FIFO queue that holds a maximum of n fragments. (Windows XP has $n = 100$ (Knockel and Crandall, 2014), and the Linux kernel in (Herzberg and Shulman, 2013) has $n = 64$). Then, every second, the attacker sends the client $n/2$ copies of its first spoofed fragment (each with different IPIDs), and $n/2$ copies of the second spoofed fragment, per Figure 2·6. Each second spoofed fragment has (1) IPID corresponding to one of the first spoofed fragments, and (2) timestamps T_2 and T_3 corresponding to the (legitimate) time that the fragment was sent plus some constant value (*e.g.,* $x = +10$ mins, where x represents how far the attacker wants to shift the client forward/backward in time). Thus, every second, a fresh batch of n fragments evicts the old batch of n fragments. The reassembled packets have offset within ≈ 1 sec, so that jitter $\psi \approx 1$ sec, and the attacker passes `TEST11`.

To surmount the second hurdle, our attack exploits the fact that IPIDs are often predictable. Several policies for setting IPID exist in the wild, including: *globally-incrementing*, *i.e.,* the OS increments IPID by one for every sent packet, *per-destination-incrementing*, *i.e.,* the OS increments IPID by one every packet sent to a particular destination IP, and *random*, *i.e.,* the IPID is selected at random for every packet (Gilad and Herzberg, 2013). Random IPIDs thwart our attacks. However, when the server uses an incrementing IPID policy, the following techniques allow our attacker to plant several copies of the spoofed fragments with plausible IPIDs (*cf.,* (Herzberg and Shulman, 2013), (Gilad and Herzberg, 2013), (Knockel and Crandall, 2014)):

Globally incrementing IPIDs: Before sending the client the $n/2$ copies of the spoofed fragments, our attacker pings the server to learn its IPID i, and sets IPID of its spoofed packets accordingly (*i.e.,* to $i+1, i+2, ..., i+n/2$).

Per-destination incrementing IPIDs: Gilad and Herzberg (Gilad and Herzberg, 2013)(Knockel and Crandall, 2014) show how per-destination incrementing IPIDs can be inferred by a puppet (adversarial applet/script that runs in a sandbox) on the client or server's machine, while (Knockel and Crandall, 2014) show how to do this without puppets. Thus, at the start of our attack, our attacker can use (Gilad and Herzberg, 2013), (Knockel and Crandall, 2014)'s techniques to learn the initial IPID i, then uses i to set IPIDs on the $n/2$ copies of its pairs of spoofed fragments. The choice of IPIDs depends on the *polling interval*, *i.e.,* the frequency at which the client queries the server. NTP default poll values range from $\tau = 6$ ($2^6 = 64$ sec) to $\tau = 10$ (1024 seconds) (Mills et al., 2010). If the attacker knew that the client was polling every 64 seconds, it could send $n/2$ copies of the spoofed fragments with IPID $i+1$,....,$i+n/2$, and then increment i every 64 seconds.

More commonly, however, the polling interval is unknown. To deal with this, the attacker can predict the IPID under the assumption that the client and server consistently used the minimum (*resp.,* maximum) polling interval, and ensures that all possible IPIDs in between are planted in the buffer. As an example, suppose that 2048 seconds (30 mins) have elapsed since the attacker learned that the IPID is i. At one extreme, the client and server could have consistently used the minimum default polling interval of $2^\tau = 64$ sec; thus, $i_{\max} = i + 2048/64 = i + 32$. At the other extreme, the client and server could have consistently used the maximum default polling interval of $2^\tau = 1024$ sec; then $i_{\min} = i + 2048/1024 = i + 2$. Then, the attacker must send pairs of spoofed fragments with IPIDs ranging from $i_{\min} = i + 2$ to $i_{\max} = i + 32$. This works as long as the fragment buffer can hold $i_{\max} - i_{\min} > 30 \cdot 2$

fragments (as in *e.g.,* Linux (Herzberg and Shulman, 2013) and Windows XP (Knockel and Crandall, 2014)). When $2(i_{\min} - i_{\max})$ exceeds the size of the fragment buffer n, the attacker repeats (Gilad and Herzberg, 2013), (Knockel and Crandall, 2014)'s techniques to learn IPID again.

Moreover, to avoid having to plant so many IPIDs in the fragment buffer, the attacker can try making the polling interval more predictable. Our experiments show that if a server becomes "unreachable" (*i.e.,* stops responding to queries) for a short period, and starts to respond with packets with *poll* field $\tau_p = 6$, the ntpd v4.2.6 client will speed up its polling interval to ≈ 64 sec. To simulate "unreachable" behavior from off-path, the attacker can plant fragments with incorrect UDP checksum (*e.g.,* planting just the second spoofed fragment, but not the first, per Figure 2·6). Then, after some time, the attack begins with *poll* set to $\tau_p = 6$ in the first spoofed fragment.

2.6.5 Conditions required for our attack.

In summary, our attack succeeds for a given victim NTP server and victim NTP client if the following hold:

1) the server accepts and acts on *ICMP fragmentation needed* packets for a 68-byte MTU, and

2) the server uses globally-incrementing or per-destination-incrementing IPID, and

3) the client reassembles overlapping IPv4 fragments as in Outcome A of Figure 2·8.

If the client reassembles packets as in Outcome B, then a weaker form of our attack could be possible, where the attacker can alter the client's time in steps of at most 1 sec. Alternatively, if the first two conditions hold and server sends NTP packets with a UDP checksum of zero, then the weak form of our attack could also be possible if the client reassembles overlapping IPv4 fragments as in Outcomes C.

2.6.6 Proof-of-concept implementation of our attack.

We implemented a proof of concept of our attack on three lab machines. Our server had per-destination incrementing IPID. We perform the small-step-big-step attack (Section 2.4.2) from off path against an ntpd v4.2.6p5 client, with a 'small step' of 10 minutes, and a 'big step' of 1 day.

Server. Our victim server ran ntpd v4.2.8p2 on Linux 3.13.0-24-generic kernel which uses a per-destination incrementing IPID. This Linux kernel has configuration parameter *min_pmtu* that determines the minimum MTU to which the OS is willing to fragment packets upon receipt of an *ICMP fragmentation needed* packet; we manually set *min_pmtu* to 68, so that the server would satisfy the first condition of our attack.[18]

Client. Choosing the right client was a challenge. It is extremely difficult to find documentation on overlapping fragment reassembly policies for popular OSes. These policies also change over time (*e.g.,* in 2005 (Novak, 2005) found that MAC OS reassembles as in Outcome A, but our July 2015 experiments indicated that MAC OS X v10.9.5 reassembles as in Outcome B). After testing various OSes, we tried using the Snort IDS to emulate a network topology where a middlebox reassembles fragmented packets before passing them to endhosts. We set up Snort in inline mode on a VM in front of another VM with the ntpd v4.2.6 client. Unfortunately, Snort's *frag3* engine, which reassembles overlapping IPv4 fragments according to various policies, exhibited buggy behavior with UDP (even though it worked fine with the ICMP fragments used in (Novak, 2005)). Finally, we gave up and wrote our own fragment buffer in python and scapy, running it on a Linux 3.16.0-23-generic OS with ntpd v4.2.6p5.

[18]The default value for *min_pmtu* in Linux exceeds 500 bytes (Schramm, 2012), so that the vast majority of NTP servers running on Linux should not be vulnerable to our attack. (Linux 2.2.13 is one notable exception; see Section 2.6.7.) However, our measurements in Section 2.6.7 indicate that servers in the wild do fragment to a 68-byte MTU; we just use Linux 3.13.0 as a proof-of-concept in our lab.

Our fragment buffer code had two parts. The first part uses scapy's sniff function to detect IPv4 fragments, and then sends them to our fragment buffer, which reassembles them and passes them back to the OS. The second part uses nfqueue to drop packets that were reassembled by the OS and pass packets reassembled by our fragment buffer. The fragment buffer itself is a FIFO queue with capacity $n = 28$ fragments and timeout $t = 30$ sec. Fragments older than t are evicted from the queue. When the queue is full, a newly-arrived fragment evicts the oldest fragment. The buffer reassembles packets according to the 'First' policy[19] in (Shankar and Paxson, 2003) (*i.e.,* Outcome A in Figure 2·8).

Attacker. Our attacker machine ran code written in scapy. Before the attack starts, we let the NTP client synchronize to the server. After that, our attacker machine should infer the IPID the server uses to send mode 4 packets to the client; rather than reimplementing the IPID-inference techniques of (Gilad and Herzberg, 2013), (Knockel and Crandall, 2014), we just have the client machine send the initial IPID i directly to the attack machine. At this point, the client no longer reveals any more information to the attacker, and the attack starts. The attacker first sends the server a spoofed *ICMP fragmentation needed* packet requesting fragmentation to a 68-byte MTU for the client; the server caches the request for 10 minutes, and starts sending the two fragments in Figure 2·7. To keep the server fragmenting, the attacker sends a fresh *ICMP fragmentation needed* every 10 minutes.

Per Section 2.6.4, each second our attacker needs to send the client a fresh batch of $n/2$ pairs of the two fragments in Figure 2·6. Each pair had IPID in $\{(i+1), ..., (i+n/2-1)\}$, with i incremented every 70 seconds.[20] Our attack machine

[19]The 'First' policy of (Shankar and Paxson, 2003) requires reassembly to prefer the fragment that was received earliest by fragment buffer.

[20]NTP uses a randomized algorithm to set the polling interval. Our client had not been running for long, so its polling interval was $\tau = 6$ (64 sec), which translates to intervals randomly chosen from the discrete set $\{64, 65, 66, 67\}$ sec. We therefore increment i every 70 seconds.However, per Section 2.6.4, (1) an off-path attacker can push the polling interval down to $\tau = 6$ by using fragmentation to make

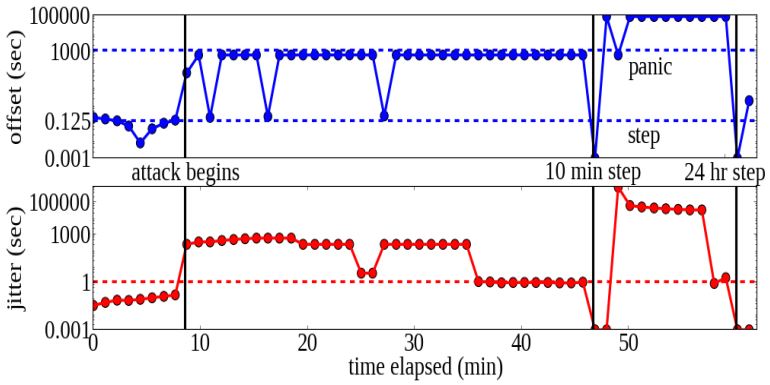

Figure 2·9: Absolute value of offset θ (above) and jitter ψ (below) computed by the client during a proof-of-concept implementation of our attack.

was a relatively less efficient eight-year old Fujitsu x86_64 with 1.8GB of memory running Linux 3.16.0-24-generic, and thus could only manage to send thirteen pairs of the required fragments within 1 second. We therefore set the size of the FIFO queue on our client machine to $n = 28$. Our attacker uses the fragmentation attack to launch a "small-step-big-step" attack : First, it sets receive and transmit timestamps T_2 and T_3 in its second spoofed fragment to shift the client 10 minutes back in time. Once the client enters 'STEP' mode, it sets T_2 and T_3 to shift the client one day back in time. (Note that an off-path attacker can check that the client is in 'STEP' mode by querying the client and checking for 'STEP' in the reference ID of the response (Mills et al., 2010, Fig. 13).)

Results (Figure 2·9). We plot the results of one run of our attack, obtained from the client's ntpq program. We plot offset θ (equation (6.2)) computed by the client for each mode 4 packet the client (thinks it) received from the server. The horizontal lines on the offset plot represent NTP's default panic threshold (1000 sec) and 'STEP' threshold (125 ms). We also plot jitter ψ (equation (5.3)) computed by the client from its eight most recent offset samples. Recall that the client will only synchronize to the

the server to look like it has become 'unreachable', and (2) if the fragment buffer has large n, our attack can accommodate larger variation in the polling interval (*e.g.,* MAC OS X has $n = 1024$ (Knockel and Crandall, 2014)).

server if $\psi < 1$ sec (Section 2.5.5, 2.6.4). Before the attack begins, the client and server are synchronized and offset is small. Once the attack begins, offset jumps to about 600 seconds (10 minutes). Figure 2·9 also shows some spikes where the offset jumps back down to a few msec. These spikes occur during cache misses, when our attacker fails to plant fragments with the right IPID in the fragment buffer; this allows the two legitimate fragments to reassemble so that the client gets sample of the correct time. The attacker pays a price each time a cache miss causes an inconsistency in the offset values; for example, at time 25 mins, the attacker crafts enough packets to force the jitter to about 10 sec, but two samples later it suffers a cache miss, causing jitter to jump up again. Eventually, the attacker crafts enough packets to keep jitter below 1 sec for some period of time, and the client accepts the time, enters 'STEP' mode, and clears its state. Once in 'STEP' mode, the attacker manages to craft nine consecutive packets, causing jitter to drop below 1 sec and sending the client back in time for another 24 hours.

2.6.7 Measuring the attack surface: Servers.

How often are the conditions required for our attack (Section 2.6.5) satisfied in the wild? We answer this question by scanning our dataset of 13M NTP servers (Section 2.3.2) to find servers satisfying the two conditions for our attack per Section 2.6.5: (1) fragmenting to a 68-byte MTU, and (2) using incrementing IPID. To avoid harming live NTP servers with this scan, we send only ICMP packets or mode 3 NTP queries (which do not set time on the server).

Fragmenting to 68-byte MTU. We send each server in our list (1) an NTP mode 3 query and capture the corresponding NTP mode 4 response, and then (2) send an *ICMP fragmentation needed* packet requesting fragmentation to a 68-bytes for NTP packets sent to our measurement machine (as per the packet in Figure 2·5, where UDP checksum is zero and IPID inside the ICMP payload is that in the captured

Table 2.6: IPID behavior of non-bad-timekeepers satisfying conditions (1), (2) of Section 2.6.5.

IPID behavior	Per-Dest $\Gamma = 1$	Globally incrementing $\Gamma = 10$	$\Gamma = 25$	$\Gamma = 50$	$\Gamma = 100$	$\Gamma = 250$	$\Gamma = 500$
# servers	2,782	5,179	2,691	533	427	135	55

mode 4 NTP response), and finally (3) send another NTP mode 3 query. If the server fragments the final mode 4 NTP response it sends us, we conclude it satisfies condition (1) for our attack.

Server IPID behavior. Next, we check the IPID behavior of each server that was willing to fragment to a 68-byte MTU. To do this, we send each IP five different NTP mode 3 queries, interleaving queries so that at about 30 sec elapse between each query to an IP. We then check the IPIDs for the mode 4 responses sent by each IP. If IPID incremented by one with each query, we conclude the server is vulnerable because it uses a per-destination-incrementing IPID. Otherwise, we determine the gap between subsequent IPIDs; if all gaps were less than a threshold Γ (for $\Gamma = \{10, 25, 100, 250, 500\}$), we conclude that the server uses a globally-incrementing IPID.

Results of server scan. Out of the 13M servers we scanned, about 24K servers were willing to fragment to a 68-byte MTU. 10K of these servers have bigger problems than just being vulnerable to our attacks: they were either unsynchronized (*i.e.,* either stratum 0 or stratum 16) or bad timekeepers (*i.e.,* with offset $\theta > 10$ sec). However, we did find 13,081 'functional' servers that fragment to a 68-byte MTU. As shown in Table 2.6, the vast majority (11,802 servers) of these are vulnerable to our attack because they use an incrementing IPIDs that grow slowly within a 30-second window. In fact, most use a globally-incrementing IPID, which is easier to attack than a per-destination IPID (see Section 2.6.4).

Who are these vulnerable servers? The majority 87% (10,292 servers) are at stratum 3, but we do see 14 vulnerable stratum 1 servers and 660 vulnerable servers

with stratum 2. Indexing these with our (very incomplete) topology data, we find that 11 servers are at the root of subtrees with over 1000 clients, and 23 servers have over 100 clients each. One vulnerable stratum 2 server, for example, is in South Korea and serves over 19K clients, another with over 10K clients is in a national provider in the UK, one with over 2K clients serves a research institute in Southern Europe, and two with over 7K clients are inside a Tier 1 ISP. Cross-referencing these servers to our May 2015 *rv* data, we find that the vast majority (9,571 out of the 11,803 vulnerable servers) are running Linux 2.2.13; the other significant group is 1,295 servers running "SunOS". We note that *not* every Linux 2.2.13 server in our *rv* dataset fragmented to a 68 byte MTU; 688 of the servers running on Linux 2.2.13 in our *rv* data responded to our final NTP query with an unfragmented NTP response, even though they had been sent a valid *ICMP fragmentation needed* packet, possibly because of middleboxes that drop ICMP packets.

2.6.8 Measuring the attack surface: Clients.

Determining how many clients in the wild satisfy the third condition of our attack (Section 2.6.5) was significantly more complex. To measure how an NTP client reassembles overlapping IPv4 fragments, we can use (Shankar and Paxson, 2003), (Novak, 2005)'s technique of sending fragmented pings. To check for reassembly per Outcome A in Figure 2·8, we send four ping fragments with offsets corresponding exactly to those in Figures 2·6 and 2·7. If the client reassembles them as in Outcome A, the reassembled ping will have a correct ICMP checksum and elicit a ping response from the client; otherwise, the client will ignore them. Figure 2·10 shows the four ping fragments and how they would be reassembled per Outcome A. We repeat this with four other ping fragments to check for Outcome B.

Before we could deploy this technique in the wild, we hit an important complication: Teardrop (BUGTRAQ mailing list, 1997), an ancient implementation bug (from

1997) with devastating consequences. In a teardrop attack, the attacker sends two overlapping IPv4 fragments to an OS, and the OS crashes. Most OSes were patched for this over fifteen years ago, but some firewalls and middleboxes still alarm or drop packets when they see overlapping IPv4 fragments. Worse yet, legacy operating systems may not be patched, and new IP stacks might have reintroduced the bug. This is a big problem for us: this measurement technique inherently requires overlapping IPv4 fragments, and thus inherently contains a teardrop attack. We therefore cannot run this measurement on all 13M NTP clients we found in the wild, since we don't know what OSes they might be running. Instead, we deal with this in two ways.

First, we have developed a website for users to measure the vulnerability of their NTP clients, *i.e.,* whether they reassemble packets as in Outcome A. (`http://www.cs.bu.edu/~goldbe/NTPattack.html`) To prevent the site from becoming a teardrop attack vector, we require users running the measurement to be on the same IP prefix as the measured NTP client.

Second, we can send our measurements to NTP clients that we know are patched for Teardrop. Teardrop affects version of Linux previous to 2.0.32 and 2.1.63 (BUGTRAQ mailing list, 1997); thus, we can use the *rv* data from the openNTPproject to determine which clients are running on patched Linux versions, and send our measurements to those clients only. We did this for 384 clients that responded to *rv* queries with "Linux/3.8.13", a kernel released in May 2013, well after Teardrop was patched. Five clients responded with pings reconstructed as in Outcome A, 51 clients with pings reconstructed as in Outcome B.

This is interesting for two reasons. Most obviously, this gives evidence for the presence of reassembly per Outcome A in the wild, which means that there are NTP clients that are vulnerable to our attack. But it also suggests that this fragmentation reassembly is not always done by the endhost itself; if it had, all 384 servers would

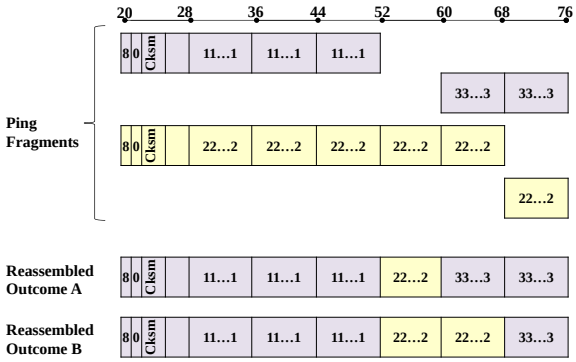

Figure 2·10: Ping packets for measuring fragmentation reassembly policies.

have responded in the same way. (Also, (Novak, 2005), (Shankar and Paxson, 2003)'s measurement indicate that Linux uses a policy that results in reconstruction per Outcome C, rather than Outcomes A or B.) Thus, we speculate that these five clients responded to our ping because they were sitting behind a middlebox that performs fragment reassembly for them

2.6.9 Recommendations: Fragmentation still considered harmful.

Our measurements suggest that the attack surface for our NTP fragmentation attack is small but non-negligible. Thousands of NTP servers satisfy the conditions required by our attack (Section 2.6.5). However, our attack only succeeds if the victim client is synchronized to a vulnerable server, and reassembles fragmented packets according to the third condition required for our attack (Section 2.6.5). Unfortunately, we could not safely measure which NTP clients reassemble packets in this way, although we do find evidence of vulnerable clients.

Perhaps the simplest way to protect the NTP ecosystem is to ensure that any OS running an NTP server does not fragment packets to a 68 byte MTU; indeed, many OSes already do this (*e.g.,* Linux (Schramm, 2012), Windows (Microsoft, 2010)). On the client side, the OS should drop overlapping NTP fragments, as should middleboxes that reassemble IPv4 fragments like (Checkpoint, 2018).

Source port randomization? One might wonder whether our attack is thwarted by

UDP source-port randomization. (That is, what if the client chose a random 16-bit value for the UDP source port (Larsen and Gont, 2011), instead of always setting it to 123?) Unfortunately, the answer is no. Suppose conditions (1), and (2) of Section 2.6.5 hold, and modify our attack as follows: replace the first spoofed fragment on the left of Figure 2·6 with an identical fragment that has its UDP source port and destination port fields sliced off. (The replacement fragment has fragment offset of 24 bytes.) Then, if the client reassembles the packet according to the 'First' policy of (Shankar and Paxson, 2003), then the packet will look just like the one reassembled per Outcome A in Figure 2·8, except with the legitimate UDP source/dest ports, and the attack will succeed. Thus, while UDP source-port randomization raises the bar for our attack, we do not consider it to be a sufficient defense.

2.7 Related work

NTP security. While NTP-based DDoS amplification attacks have generated widespread interest (see *e.g.,* (Czyz et al., 2014a)), there is less research (Mizrahi, 2012b; Selvi, 2014; Klein, 2013; Mills, 2011; Corbixgwelt, 2011) on the implications of shifting time via NTP attacks. A few researchers (Mizrahi, 2012b; Selvi, 2014; Mills, 2011; Selvi, 2015) have considered the implications of attacks on NTP traffic, and (Selvi, 2014; Selvi, 2015) demonstrated on-path attacks on timing clients that update their clocks at predictable intervals. We consider on-path attacks on the full NTP implementation (ntpd), which does *not* perform clock updates in a predictable fashion, and present new off-path attacks. Complementary to our work are efforts to identify software bugs in ntpd (National Vulnerability Database, 2014), (Röttger, 2015), including several that were made public at the same time as our work (Chiu, 2015); because ntpd typically operates as root on the host machine, we expect that interest in this area will only continue to increase. Our work is also related to older NTP measurement

studies (Minar, 1999), (Murta et al., 2006), as well as the recent work of (Czyz et al., 2014a); (Czyz et al., 2014a) studies NTP's use for DDoS amplification, but we focus on the integrity of timing information. Finally, concurrent to our work is a study that measures network latency using NTP (Durairajan et al., 2015).

IPv4 Fragmentation. Our work is also related to research on exploiting IPv4 packet fragmentation for *e.g.,* off-path attacks on operating systems (BUGTRAQ mailing list, 1997), DNS resolvers (Herzberg and Shulman, 2013), TCP connections (Gilad and Herzberg, 2013; Gont, 2010), and to evade intrusion detection systems (Ptacek and Newsham, 1998), (Shankar and Paxson, 2003), (Novak, 2005), (Baggett, 2012) and exploit side channels (Knockel and Crandall, 2014). Unlike most prior work, however, we had to use fragmentation to craft a *stream* of self-consistent packets, rather than a single packet. Our attack also exploits problems with overlapping IPv4 fragments (Novak, 2005), (Shankar and Paxson, 2003), (Baggett, 2012) and tiny IPv4 fragments, and should provide some extra motivation for OSes/middleboxes to drop tiny/overlapping fragments, rather than reassemble them.

2.8 Conclusion

Our results suggest four ways to harden NTP:

1) In Section 2.4 we discuss why freshly-restarted ntpd clients running with the `-g` configuration (which is the default installation for many OSes) are vulnerable to quick time shifts of months or years. We also present a 'small-step-big-step' attack, captured in CVE-2015-5300, that allows an on-path attacker to stealthily shift time on a freshly-restarted ntpd client. Different versions of the small-step-big-step attack succeed on ntpd v4.2.6 and ntpd v4.2.8. To protect against these attacks, users can either stop using the `-g` configuration, or monitor their systems for suspicious reboots of the OS or of ntpd. Section 2.4.3 also has recommendations for implementors who

wish to patch against our small-step-big-step attack.

2) We showed how NTP's rate-limiting mechanism, the Kiss-o-Death packet (KoD), can be exploited for off-path denial-of-service attacks on NTP clients (Section 2.5.3). We find that ntpd versions earlier than v4.2.8p3 allow for trivial spoofing of KoD packets, and that this can be exploited to disable NTP on clients in the Internet using only a single attacking machine. The KoD-spoofing vulnerability (CVE-2015-7704) has been patched in ntpd 4.2.8p4, following the disclosure of our work. This patch, however, does not prevent off-path attackers from *eliciting* valid KoDs from the server via a priming-the-pump technique (CVE-2015-7705); the priming-the-pump technique, however, does require the attacker to expend more resources (*i.e.,* send more packets). Therefore, we argue in Section 2.5.7 that NTP should either (1) eliminate its KoD functionality, (2) require NTP clients to cryptographically authenticate their queries to NTP servers, or (3) adopt more robust rate limiting techniques, like (Vixie, 2014).

3) In Section 2.6 we showed how an off-path attacker can use IPv4 fragmentation to hijack an NTP connection from client to server. Because our attack requires server and client to run on operating systems that use less-common IPv4 fragmentation policies, we have used a measurement study to quantify its attack surface, and found it to be small but non-negligible. As we argue in Section 2.6.9, NTP servers should run on OSes that use a default minimum MTU of ≈ 500 bytes, as in recent versions of Linux and Windows (Schramm, 2012), (Microsoft, 2010). OSes and middleboxes should also drop overlapping IPv4 fragments. We have also set up a website where operators can test their NTP clients for vulnerability to our fragmentation attacks.[21]

4) Each of our attacks has leveraged information leaked by the *reference ID* field in the NTP packet (Figure 2·1). (In Section 2.4, we use the reference ID to learn that the client was in the 'INIT' or 'STEP' state. In Section 2.5.3 our off-path attacker

[21] `http://www.cs.bu.edu/~goldbe/NTPattack.html`

used the reference ID to learn the IPs of client's servers.) Moreover, since the purpose of the reference ID is to prevent timing loops (where A takes time from B who takes time from A (Stenn, 2015c)) any extra information leaked in the reference ID should be limited.

In fact, RFC 5905 (Mills et al., 2010, pg 22) already requires IPv6 addresses to be hashed and truncated to 32 bits before being recorded as a reference ID. Of course, this approach is vulnerable to trivial dictionary attacks (with a small dictionary, *i.e.,* the IPv4 address space), but there are other ways to obfuscate the reference ID. One idea is to use a salted hash, in an approach analogous to password hashing. Upon sending a packet, client A chooses a a fresh random number as the 'salt', includes the salt in the NTP packet (perhaps as the lower-order bits of the reference timestamp (Figure 2·1)), and sets the reference ID in the packet to Hash(IP, salt), where IP is the IP address of the server B from which A takes time. B can check for a timing loop by extracting the salt from the packet, taking B's own IP, and checking that Hash(IP, salt) matches the value in the packet's reference ID; if yes, there is a timing loop, if no, there is not. This approach requires no state at A or B, and de-obfuscating the IP requires an attacker to recompute the dictionary for each fresh salt. This approach, however, comes with the caveat that an committed attacker could still launch dictionary attacks on the salted hash.

5) Our attacks also exploited the fact that, by default, ntpd sends mode 4 responses in response to any mode 3 query sent by any IP in the Internet. Thus, it makes sense to limit the set of IPs that can communicate with an NTP client, especially for clients that are not designed to serve time to the Internet at large. (My laptop, for example, should not be responding to NTP queries from anyone.) An organization can also run its entire NTP infrastructure (including stratum 1 servers) behind a firewall, to avoid attacks from the public Internet.

Our work also motivates taking another look at cryptographic authentication for NTP (Haberman and Mills, 2010), (Mizrahi, 2012b), (Röttger, 2012), (Franke et al., 2018).

Chapter 3

Attacking NTP's Authenticated Broadcast Mode

3.1 Introduction

The Network Time Protocol (NTP) (Mills et al., 2010), one of the Internet's oldest protocols, is used to set time on Internet clocks. Time places a crucial and often-ignored role in the security and correctness of computing applications, and especially in cryptographic protocols. As we discussed in Chapter 2 (Section 2.2), an attacker that manipulates time using NTP can seriously undermine the security of key Internet protocols and applications, including TLS certificates (Selvi, 2015), (Mills, 2011), (Klein, 2013), DNSSEC, routing security with the RPKI, authentication with Kerberos, caching, and bitcoin (Corbixgwelt, 2011). NTP operates in several modes including (1) client/server, (2) symmetric active/passive, and (3) broadcast/multicast. Our earlier work (Chapter 2) considered attacks on NTP's client/server mode. In this companion Chapter, we consider the security of NTP's broadcast mode.

We use network measurements to find that NTP's broadcast mode, which is intended for an environment with a few servers and potentially a large client population, is used by thousands of NTP clients in the wild (Section 3.5). Next, we show that while symmetric-key cryptographic authentication of NTP broadcast traffic is recommended by the NTP specification (Mills et al., 2010) and required by the open-source

NTP reference implementation *ntpd*, it does not provide sufficient protection against attacks on broadcast mode. We consider both (1) on-path attacks, where the attacker occupies a privileged position on the path between NTP client and one of its servers, and (2) off-path attacks, where the attacker can be anywhere on the network and does not observe the traffic between client and any of its servers. We present an on-path replay attack on authenticated broadcast mode (CVE-2015-7973) that causes the NTP client to get stuck at a particular time (Section 3.3), and a new off-path denial-of-service attack on authenticated broadcast mode (CVE-2015-7979) that also applies to all of NTP's "preemptable" and "ephemeral" modes of operation (Section 3.4). We conclude by discussing the inherent challenges of cryptographically authenticating NTP's broadcast mode, and provide several recommendations for the way forward (Section 3.6).

3.2 NTP's broadcast mode

NTP clients and servers are not configured to operate in broadcast mode by default on most operating systems. However, there is a configuration option that allows for this mode of operation.

Broadcast servers. An NTP broadcast server can be preconfigured to periodically send '*persistent*' broadcast-association server packets (*NTP mode 5 packets*) to the clients on the broadcast network. By *persistent*, we mean the server mobilizes the broadcast association upon initialization, and never demobilizes the association (Mills et al., 2010). Figure 3·1 presents a sample NTP mode 5 broadcast packet.

Broadcast clients. An NTP client can be preconfigured to accept NTP mode 5 packets.[1] When a broadcast client receives its first NTP mode 5 packet, the client

[1]The configuration option *broadcastclient* or *multicastclient [address]* allows an ntpd client to receive and process mode 5 broadcast packets. Note that a client configured to accept multicast messages from a particular address also accepts broadcast messages from ANY address.

must first calculate the propagation delay by exchanging a volley of client/server mode packets with the broadcast server—where the client sends the server an NTP mode 3 query and the server responds with an NTP mode 4 response.[2] After this, the client reverts to broadcast client mode, and creates an *ephemeral* association with the server upon receipt of further mode 5 broadcast packets. An *ephemeral* association is mobilized upon arrival of a packet and exists until error or timeout (Mills et al., 2010).

Authenticating an association. How does an NTP client validate incoming packets before establishing an association with a server? Most NTP traffic (especially client/server-mode traffic) is not cryptographically authenticated.[3] However, even in the absence of cryptographic authentication, NTP clients running in client/server mode or symmetric active/passive mode use a *nonce* to validate a server's response. The nonce is a field in the NTP packet, called the *origin timestamp*; see Figure 3·1. Upon receipt of an NTP response packet, the client checks if the 64-bit *transmit timestamp* field in the most recent query packet it sent the server, matches the 64-bit *origin timestamp* field in the incoming response packet. This is called *TEST2* in the NTP specifications. This non-cryptographic authentication is based on the premise that the nonce has enough entropy such that an *off-path* attacker, who can not see the NTP packets in transit, cannot guess the nonce. Indeed, as we argued in Chapter 2, we can safely assume that this nonce has about 32 bits of entropy, and so it is difficult

[2]The server and client also run the Autokey security protocol, if they are configured to do so. Autokey (Haberman and Mills, 2010) is public-key authentication method for NTP, but NTP clients do not request Autokey associations by default (Autokey, 2012), and many public NTP servers do not support Autokey (*e.g.,* servers in `pool.ntp.org`). In fact, a lead developer of the ntpd client wrote in 2015 (Stenn, 2015a): "Nobody should be using autokey. Or from the other direction, if you are using autokey you should stop using it." We therefore do not consider Autokey any further here.

[3]As we discussed in Chapter 2, NTP's symmetric-key cryptography is not commonly because the symmetric keys must be pre-configured manually ; this can be quite cumbersome for public servers that must accept queries from arbitrary clients. (NIST, for example, distributes symmetric keys for its public servers via US mail or facsimile (NIST, 2010).) Moreover, NTP's public-key cryptography (Autokey) is not recommended for use in the wild, see footnote 2.

to forge from off path.

Authenticating broadcast. In contrast to the client/server mode, where the client actively sends a query to the server to get the response, the broadcast client operates in *listen-only* mode. Thus, because the client does not send the server any queries for broadcast packets, and the *origin timestamp* field in the broadcast server packet is always set to *null*. So now *TEST2* that defends NTP's client-server mode from off-path attacks does not apply. Moreover, the NTP's current reference implementation (ntpd v4.2.8p4) does NOT use UDP source port randomization (Larsen and Gont, 2011), and so an off-path attacker can easily forge an unauthenticated mode 5 packet.

Also, an NTP client preconfigured to run in broadcast client mode will accept and process packets from ANY server that sends it broadcast packets; it is NOT configured to listen only to one particular broadcast server. So any off-path attacker can easily send broadcast messages and the client will accept them. RFC5905 (Mills et al., 2010, pg 57) says:

> Filtering can be employed to limit the access of NTP clients to known or trusted NTP broadcast servers. Such filtering will prevent malicious traffic from reaching the NTP clients.

To fill this gap and the lack of nonce check, RFC 5905 (Mills et al., 2010) strongly suggests the use of cryptography to authenticate broadcast packets. Indeed, NTP's current reference implementation (ntpd v4.2.8p4) requires *symmetric-key* cryptography, by default, for clients that wish to listen to broadcast mode packets. NTP's symmetric cryptographic authentication appends an MD5 hash keyed with symmetric key k of the NTP packet contents m as MD5($k||m$) (Mills, 2011, pg 264) to the NTP packet in Figure 3·1; authenticated NTP packets also have a 32-bit *key ID* which is used to identify the symmetric key that was used to authenticate the message.

In this paper, however, we present two attacks that show that NTP's symmetric

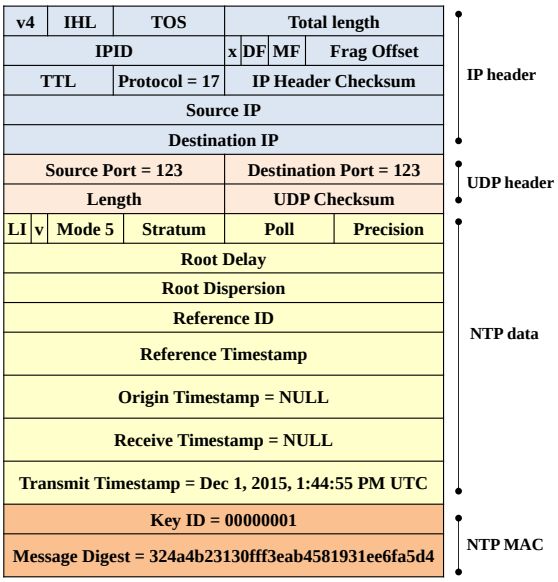

Figure 3·1: Mode 5 NTP Broadcast Packet.

key cryptography does not provide sufficient protection for broadcast mode.

3.3 Timeshifting Attacks

Should broadcast be robust to replay attacks? According to RFC 5905 (Mills et al., 2010), NTP's "on-wire protocol ... resists replay of a server response packet." This is supposed to be accomplished through what is called *TEST1* in the protocol specification: Upon receipt of an NTP response packet, the NTP client matches the transmit timestamp in the current packet to that of the last response packet it received; if the timestamp matches, it marks the packet as duplicate and drops it. However, we now show that because broadcast mode does not impose TEST2, then TEST1 cannot provide sufficient protection against replay attacks, even when NTP packets are cryptographically authenticated.

a) Deja Vu: Our on-path time-sticking attack (CVE-2015-7973). Consider a man-in-the-middle (MiTM) attack, where the attacker is positioned between the server and the victim client, and can intercept and replay a packet and prevent onward

transmission of the original packet, but does not possess the symmetric key that authenticates broadcast messages. We show that the protocol does not resist the following replay attack. The MiTM collects and records a contiguous sequence of server broadcast packets. (The attacker requires a sufficient number of these packets for the client to update its clock; this is because NTP requires a client to obtain between eight to hundreds of messages from a server before the client's *clock discipline algorithms* synchronizes it to the server (Mills et al., 2010, Sec. 10-12).) He then replays this sequence of packets, over and over, to the victim client; the victim accepts the same time over and over, and thus gets stuck at a particular time. Notice that these are the authenticated packets from the broadcast server, and so they pass the authentication check on the client. Moreover, by replaying a *sequence* of packets, rather than just one packet, the attacker ensures that the replayed packets pass *TEST1*.

b) Our off-path time-shifting attack. If the attacker is one of the clients on the broadcast network, or on an adjacent network that also gets the broadcast packets from the same broadcast server, it then shares the same symmetric key with the server as the victim client. In this case, the attacker possesses the same key as the server and therefore can simply forge authenticated NTP mode 5 packets and send them to the victim client. The attacker can then send the victim back/forward in time as discussed in Chapter 2, or can make him stick to a particular time.

Why do these attacks work? These attacks highlight the following weaknesses in NTP; a) The protocol specifies and defaults to the use of symmetric key cryptography for broadcast authentication, where all nodes share the same key and one/some of them could be malicious or may be compromised, b) an NTP client is unable to recognize that it is stuck in a particular time for long periods of time, and c) in the absence of *TEST2*, *TEST1* doesn't actually prevent replay in general—it just

prevents replay of the most recent packet. Our replay attack passes *TEST1* because the client only matches the current transmit timestamp with that of the last packet.

Experiments: As a proof-of-concept, we set up an ntpd v4.2.8p3 broadcast client and server using the configuration options *broadcastclient* and *broadcast IP_address_range*. Another machine on the same network behaves as MiTM and collects 12 *mode 5* packets and stores them. The MiTM then drops the original *mode 5* packets to the victim and replays his previously collected set of *mode 5* packets. The victim accepts the time after getting sufficient samples required for a server to pass the clock discipline algorithms, gets into the 'STEP' mode[4]and clears the state variables for this association. We continue this experiment for ≈ 4 hours and observe that the victim's system clock is stuck at the same time.

Implications of the attack. A MiTM can use a replay attack to make the victim client get stuck at a particular time value forever. Moreover, a compromised machine on the same or adjacent subnet can or forge authenticated *mode 5* packets and shift time forward or backward on the victim client. Shift time forward/backward has severe implications on security guarantees provided by various core Internet protocols, such as DNSSec, BGP, TLS, and authentication services that use Kerberos; see Chapter 2 for discussion.

3.4 Denial of Service Attacks

We now present an off-path denial-of-service attack that generically succeeds on any preemptable or ephemeral association that is cryptographically authenticated, including authenticated broadcast mode.

Preemptable and empheremal associations. NTP's broadcast clients use an

[4]An NTP client enters 'STEP' mode whenever it needs to shift its clock by more than 125ms but less than $\approx$ 16 min; our replay attack shifts the client back in time by more than 125ms, causing the client to enter STEP mode.

ephemeral association to listen to NTP mode 5 from a broadcast server; as discussed in Section 3.2, this association is automatically demobilized upon error or timeout.

NTP also supports *preemptable* associations (Mills, 2014), which are similar to ephemeral associations. Preemptable associations are mobilized if the ntpd client has the keyword "preempt" to the line in its configuration file that establishes a association with a particular server. Alternatively, the ntpd client may be preconfigured with the *manycastclient* or *pool* [*pool_address*] options; in this case, the client establishes a preemptable association upon receipt of a server discovery packet. Preemptable associations are also demobilized upon error or timeout.

Our off-path DoS attack (CVE-2015-7979). An off-path attacker can easily cause an error by sending *mode 5* with bad cryptographic authentication (*e.g.,* wrong key, mismatched key, incorrect message digest, *etc.*). The attacker sends one such error-causing packet for every legitimate response the client receives from the server, so that the client immediately tears down its association with the server. This way, the client never collects enough good NTP response to allow its clock discipline algorithms to update its local clock, resulting in a denial-of-service attack on the client.

Experiment. As a proof-of-concept, we set up ntpd v4.2.8p3 broadcast client and server using the configuration options *broadcastclient* and *broadcast IP_address_range* respectively. Once the client is synchronized with the broadcast server, another machine which behaves as an off-path attacker sends badly-authenticated *mode 5* packet to the client. The client immediately tears down the association with the server and clears all the state variables. Next, the client receives the legitimate packet from the broadcast server and again mobilizes the association. The attacker again sends the bad *mode 5* packet and the client again tears down the association. The attacker keeps repeating this and the client never obtains enough consistent time samples from the server to allow it to update its system clock.

Implications a) An off-path attacker can deny NTP service to the broadcast client even when it uses cryptographic authentication. b) If the client is preconfigured to a bad timekeeper or one of the servers' that the client is configured to is controlled/compromised by the attacker, then using this DoS attack, the client can pin the client to bad server that is controlled by him. The attacker can then send the client back/forward in time which has implications as mentioned in Section 3.3.

3.5 Measurement Results

We use NTP's *peers* command to check for the presence of broadcast and other ephemeral and preemptable modes in the wild. As shown in Figure 3·2, NTP's *peers* command returns a list of all associations used by an NTP client; associations with a broadcast server are marked with a *b* or *B*, client/server associations are marked with a *u*, *etc.*. '*' is used to indicate the association that the client last took time from, and '+' indicates an association that is a candidate for synchronization. While this command provides a variety of useful information for network measurement, it's also a great tool for adversarial network reconnaissance and DDoS amplification attacks (Czyz et al., 2014a); for this reason, network operators commonly disable remote *peers* queries, or configure firewalls or other middleboxes to drop them. Moreover, while we conjecture broadcast mode is most common when both the clients and the broadcast server are behind a NAT, we are unable to scan clients behind a NAT. Therefore, it's important to remember that our measurement results can only provide a lower bound on the number of broadcast/ephemeral/preemptable associations in the wild.

Our scan. Thus, we scanned the entire IPv4 address space using the *peers* command on 10-11 November 2015, and obtained responses from 4,443,118 IPv4 addresses. On 16-21 November 2015 we rescanned only the 4.4M responding IP addresses with NTP's

peers command, as well the *rv* command (which reveals useful information about the NTP client, including its version, build date, and the operating system it runs on), and the *as* command (which has useful information about each association used by the client). For this second scan we obtained responses from 3,716,362 IPv4 addresses; we consider only these addresses here.

Results. Of the 3.7M responding IPs, we found that 18,020 (0.4%) IPs have at least one broadcast association, and 1,767 IPs use multicast associations. (Recall that clients configured for multicast will also accept broadcast associations.) Moreover, we see 9,806 IPs that use broadcast associations exclusively, of which 7,556 IPs were synchronized to a broadcast server, while the remaining 2,250 were unsynchronized and thus likely malfunctioning.

As an aside, we were also surprised to find many symmetric associations in the wild; 190,724 (5.1%) of the responding IPs had at least one symmetric association. Overall, we found 2,848,238 IPs that have at least one client/server associations, 77 IPs use multicast exclusively, 67383 IPs use symmetric associations exclusively, and 9806 use broadcast exclusively. This is a total of 2,925,504 (78.7%) IPs; for the rest of the IPs, their association status is "-" which may mean they are initializing, or using a local clock (*e.g.,* via GPS) rather than taking time from the Internet using NTP. Thus, while broadcast is not an especially popular mode of operation for NTP, we do find thousands of clients in the wild that rely upon it for time synchronization.

Who are these broadcast clients? Of the 18K IPs that have at least one broadcast association, 16,552 of them also responded to NTP's *rv* query, and thus reveal information about their operating systems, ntpd version, and compile date. Most of these broadcast clients are running on unix (10,671 IPs) or 'cisco' devices (5,135 IPs). Also, out of 16.5K IPs that responded to the *rv* query, only 326 replied with the detailed ntpd version and compilation details (the rest merely say "ntpd version 4").

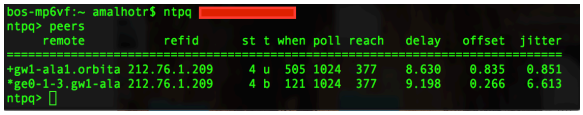

Figure 3·2: Sample response to *peers* query.

Of these, the majority (212 IPs or 65%) of these have been compiled between 2012 and 2015 inclusive. The most popular ntpd version that we found is 4.2.6p5@1.2349 (23%) which was released in December 2011, closely followed by 4.2.8@1.3265 (20%) which was released in December 2014, while 4.1.1c-rc1@1.836 (released in 2001) and 4.2.4p5-a (released in 2008) are 9% each. The bottom line is that we do find evidence of recently maintained NTP implementations that use broadcast mode in the wild.

3.6 Recommendations

We have several recommendations to mitigate our attacks.

1) Ephemeral and pre-emptable associations considered harmful. Our denial-of-service attack from Section 3.4 points to a serious problem with the notion of ephemeral and preemptable associations; namely, that an off-path attacker can easily forge a packet that can tear down an association. Even though an ephemeral association can easily be reestablished, the attacker can quickly tear it down again before the client has the chance to update its clock. For this reason, we suggest that NTP does NOT tear down ephemeral associations upon receipt of a malformed packet; instead, the malformed packet should just be dropped, while the association remains in place.

2) Prevent replay in broadcast mode. As others have pointed out (Mizrahi, 2012b), (Franke et al., 2018), NTP's broadcast mode should contain a robust mechanism for preventing replay attacks. TEST1 is insufficient, since it only checks if the most recent packet has been replayed. One solution is to require authenticated mode 5 NTP packets to include an incrementing counter (*e.g.,* in the extension field). Another idea is to use the transmit timestamp field in the mode 5 response packet

(Figure 3·1) as an incrementing counter; to do this, the broadcast client would need to ensure that the transmit time is monotonically increasing. Alternatively, the MAC computed on the broadcast packet could become a hash chain; that is, the MAC on packet p_i should be computed over the contents of packet p_i concatenated with the MAC on packet p_{i-1}.

3) Monitor to detect time-sticking. In the absence of replay protection, monitoring could be used as a "band-aid" solution against time-sticking attacks. That is, NTP clients could monitor their own clocks to see if they are stuck at the same timestamp for a considerable amount of time; if so, they could log an error or alert to warn an admin.

4) Only the broadcast server should be able to sign broadcast packets. As others have pointed out (Mizrahi, 2012b),(Franke et al., 2018), the broadcast servers' symmetric key should NOT be distributed to all its client; as we noted in Section 3.3, this allows clients to trivially forge packets from the server. Instead, the only entity that should be able to sign broadcast (mode 5) packets is the broadcast server itself.

3.7 Conclusion

Our analysis highlights the difficulty of designing robust cryptographic authentication for NTP's broadcast mode. One might be tempted to dismiss broadcast mode altogether, by arguing that broadcast mode is a legacy from the past that is largely unused today. Our measurements, however, indicate that this is not the case; while broadcast mode is not especially popular, we do find thousands of NTP clients in the wild that have broadcast associations. Thus, we believe that the community should take another careful look at authentication for NTP's broadcast mode.

One approach, taken by a new Internet draft for the "Network Time Security protocol (NTS)", achieves recommendations (2) and (4) above through a modified version

of TESLA (Perrig et al., 2005). TESLA uses public-key cryptography to ensure that a server is the only entity that can authenticate broadcast messages, but that the authenticators themselves can be computed and validated using fast symmetric cryptography. To do this, however, TESLA requires loose time synchronization between the broadcast server and its client. Thus, using TESLA in the context of NTP creates a circular dependency on time. NTS suggests avoiding this circular dependency by using an authenticated unicast association to achieve the loose synchronization between the server and each of its clients. This, however, once again requires pairwise associations between server and each client, and may defeat the purpose of using the broadcast mode in the first place. Finding a cryptographic solution that can authenticate NTP's broadcast mode, without a unicast association or a circular dependency on time, remains an interesting open problem.

Chapter 4

The Impact of Time on DNS Security

4.1 Introduction

Time is a crucial building component of network protocols, providing basic correctness and functionality as well as claimed security guarantees. Recently, however, it has become clear that network protocols cannot take the correctness and security of time for granted. There have been a series of works describing errors, misconfigurations and malicious attacks that are possible when time is obtained from network timing protocols such as the Network Time Protocol (NTP) and Simple NTP (SNTP), (Malhotra et al., 2016), (Selvi, 2014), (Malhotra and Goldberg, 2016), (Malhotra et al., 2017), (Czyz et al., 2014b). It is known that millions of hosts (Czyz et al., 2014b), (Malhotra et al., 2016), (Mauch, 2015), (Minar, 1999), (Murta et al., 2006) on the Internet implement and run network timing protocols. However, there is little work exploring how a reliance on network timing protocols affects the security of other network protocols that rely on time. (Thus far, there is only the work of Selvi on SNTP's impact on the security of HSTS (Selvi, 2015).) In this work, we explicitly look into how a reliance on time affects the security of the Domain Name System (DNS) and DNSSEC. We show how attacks on time and NTP can be used as a pivot for attacks on DNS and DNSSEC.

DNS relies on caching to provide enhanced performance and improved reliability in the face of network failures. Our work considers how time can be exploited to attack the correctness and security of caching, and to expose the DNS to Denial of

Service (DoS) and cache poisoning attacks. We also consider how the DNS security extensions (DNSSEC) can be attacked by exploiting time.

Time can be represented as absolute time (*e.g.,* "midnight August 7, 2018") or relative time (*e.g.,* "20 minutes from the moment you read this"). While the relative time is typically obtained from internal sources (*e.g.,* oscillators, CPU timers), absolute time is typically obtained from an external source (*e.g.,* manual settings, network timing protocols like NTP). As we have seen from recent work (Selvi, 2014), (Malhotra et al., 2016), (Malhotra and Goldberg, 2016), (Malhotra et al., 2017), this reliance on external time sources leads to vulnerabilities.

DNS and Time The DNS protocol's reliance on time comes through its use of Time To Live (TTL) values for DNS records. Fortunately, the DNS on-the-wire protocol specification uses only relative time values for the TTL, and thus is not directly attackable. However, DNS caching implements TTL as an absolute time value, and is therefore attackable. This issue is not just the result of an implementation flaw; we point out that the earliest DNS RFCs (RFC 1035) implicitly assume that DNS caches rely on absolute time (Section 4.2.3). Moreover, the DNSSEC on-the-wire protocol also relies on absolute time to determine the validity interval for DNSSEC signatures; thus, we show DNSSEC is also vulnerable to attacks that pivot from time.

Attacks & Implications To support these claims, we show two concrete attacks that leverage absolute time to attack DNS resolver caches. First, we present a *cache expiration attack*: when time is shifted forwards, the DNS cached responses expire sooner than expected, effectively flushing the cache. Second, we present a *cache sticking attack*: when time is shifted backwards, the cached responses stick in the cache for longer than intended. We show how these attacks can be used to harm DNS performance (introducing latency into DNS responses) and DNS availability (increasing the risk of denial of service). We also discuss how they can be used to aid

for fast-fluxing, cache poisoning and other well-known threats to the DNS. We also present two similar attacks on DNSSEC signature validation; by shifting the time on the validating resolver forwards or backwards, we can force a valid signature to be deemed invalid, causing DNSSEC denial-of-service and several other problems.

Measurements How big is the attack surface? While there are many ways to attack the absolute time, in this work we specifically focus on NTP as an attack vector. In Section 4.4, we use DNS and NTP measurements to see how many DNS resolvers might be vulnerable to our cache expiration and cache sticking attacks via pivots from NTP. We distinguish between two types of attacks on NTP. (1) In an on-path attack on NTP, the attacker holds a privileged position on the network between a victim system and its NTP timeserver. (2) In an off-path attack on NTP, the attacker does *not* sit between the victim system and it's NTP timeserver; the off-path attacker need only have the ability to spoof NTP packets (which are sent via UDP, the transport protocol that is used by NTP). Naturally, off-path attacks are "a scarier threat", because the attack can be any remote machine on the Internet.

We measure both open resolvers identified by the Open Resolver Project (Mauch, 2018), and private (non-open) resolvers that are accessible via RIPE Atlas Probes (ripe, 2018). We find a significant attack surface. At least 33% of our measured private DNS resolvers and 5% of our measured open resolvers are vulnerable to MiTM attacks on NTP. Meanwhile, 6% of those (33%) private DNS resolvers and 19% of those (5%) open resolvers are vulnerable to "even scarier" off-path attacks on NTP (Malhotra et al., 2017), many of which should have been patched years ago. (Indeed, one of the relevant patches was shipped in ntp-4.2.8p6, released on 19 January 2016.)

Recommendations & Implementation Our attacks and measurements indicate that DNS should not continue to rely on the correctness of absolute time (which is often derived from NTP). Instead, we suggest using the *raw time* or *adjusted raw*

time (Section 4.2.1) as operating-system obtained sources of relative time for DNS caching. Meanwhile, for DNSSEC signature validation, resolvers are doomed to use the vulnerable absolute time. The only way to secure DNSSEC against timing-related attacks is to secure the aboslute time (*e.g.,* to secure NTP).

Upon our disclosure and recommendations, we worked with the open-source implementation of the Unbound resolver (Unbound, 2018) to built a prototype implementation that uses raw time (rather than absolute time) for DNS caching. Unbound is committed to release this update in the coming version of Unbound. (See section 4.5.1 for details on implementation.) Also, after a preliminary disclosure of our work at IETF 100, the knot resolver (knot, 2018) co-incidentally also changed how it used time for DNS caching. These changes were released in knot v1.5.1. In Section 4.5.1 we review their changes and discuss the extent to which they actually harden knot against our attacks. (As a spoiler, we find that these changes are not sufficient to completely stop our caching attacks on DNS.)

4.2 Time and the DNS ecosystem

In this section, we give background on why attacks on time can be used to harm the correctness of DNS. We start with an overview of absolute time and relative time (Section 4.2.1) and then discuss how the DNS specifications and implementations treat time (Section 4.2.3).

4.2.1 Absolute time vs Relative time

Protocols and applications can express time in several forms, depending on whether or not universal agreement is required about that point in time. This section focuses on the differences between absolute time and relative time.

Absolute Time Absolute time expresses an absolute point in time (*e.g.,* June 13, 2018 at 1:32:09pm).For instance,"Unix Time" is seconds since midnight January 1st,

1970, while "Universal Coordinated Time" (UTC) is an international time scale that forms the basis for the coordinated dissemination of standard frequencies and time signals (BIPM, 2018). Absolute time is often used to express the validity of objects with a limited lifetime that are shared over the network[1]. In order to validate absolute time value, a system needs access to a reasonably close reference time, for instance one based on the UTC.

How do systems get absolute time and why is it vulnerable? The absolute time on a system is known as *system time*. One way to update system time is to manually enter the date and/or time. One can also set the system time from the local machine by using the hardware time, which is maintained by a battery-powered clock that persists upon reboot. Alternatively, one can get the absolute time from the Internet, using a variety of timing protocols including the Network Time Protocol[2] (NTP) (Mills et al., 2010), chronyd (chrony, 2015), sntpd (Mills, 2006), openNTPD (openNTPD, 2012) and others.

There are several problems with relying on system time. First, manual configurations can be subject to errors and misconfiguration. Also, for some machines, when moving between time zones, the system time must be corrected manually. Second, because accessing the hardware time requires an I/O operation which is resource intensive, many systems use hardware time *only* upon reboot, to initialize the system time; subsequent updates to the system time are made either manually or through NTP (Linux, 2018), (gentoo linux, 2018), (time, 2018). However, systems like microcontrollers that operate within embedded systems (*e.g.,* Raspberry Pi, Arduino, *etc.*) often lack internal hardware to keep track of time. When embedded systems require synchronization with the absolute time, they typically initialize their base time upon reboot by obtaining the current time from an external source (*e.g.,* a timeserver or

[1]For instance, PKIX certificates (Cooper et al., 2008) carry two time values expressing their earliest and latest validity.

[2]NTP disseminates UTC time.

an external clock), or by asking the user to manually enter the current time (wiki, 2018). Third, relying on Internet timing protocols opens up the system time to attack. Recent papers show vulnerabilities in NTP (Malhotra et al., 2016), (Malhotra et al., 2017), (Malhotra and Goldberg, 2016) and SNTP (Selvi, 2014) that allow attackers to maliciously alter system time - pushing system time into the past or even into the future. Moreover, many of these *time-shifting* attacks can be performed by *off-path attackers*, who do not occupy a privileged position on the network between the victim system and its time sources on the Internet. Researchers have also demonstrated off-path *denial of service attacks* on timing protocols that prevent systems from synchronizing their clocks. The bottom line is that obtaining system time from an external sources create dependencies that can be exploited.

Relative Time *Relative time* measures the time interval that has elapsed from some reference point (*e.g.,* "20 minutes from the time of your query"). Relative time is commonly used in network protocols, *e.g.,* to determine when a packet should be considered "dropped", or *e.g.,* to set *Time To Live (TTL)* values that govern the length of time for which an object is valid or usable. Relative time does *not* require access to the UTC time, or any other absolute time metric—only the rate of passage of this time across different systems is important.

How do systems get relative time? The relative time on an operating system can either be a *raw time* or an *adjusted raw time*. In both cases, relative time is monotonic. Importantly, the key property of each type of time source is not its current value, but rather the guarantee that the time source is monotonically increasing and thus useful for calculating the difference in time between two points (Love, 2013).That said, there are several caveats to each type of time source.

Raw time At its most fundamental, a system has its own perception of time; its unmodified, *raw time*. This time is typically measured by counting cycles of an

oscillator, but systems can also use process CPU time or thread CPU time (via timers from the CPU). The quality of the raw time is therefore dependent on the either the stability of the oscillator or of the CPU timer. Raw time is a purely subjective time—no general meaning can be attached to any specific value. One can only obtain the relative time by comparing two values. Because raw time is unaltered by any external or manual time source, the raw time is continuous and strictly monotonically increasing; it always increases, never decreases, never makes unexpected jumps, and never skips. It is not subject to vulnerabilities or dependencies in external time sources. Importantly, even if highly accurate oscillators are used, raw time passes at a slightly different rate than system time. This difference is called clock drift. Raw time is not adjusted for the error introduced by clock drift. Thus, the accuracy of raw time is dependent on the clock drift, which further depends on factors including oscillator quality, system load, or ambient temperature, *etc.*.

Adjusted raw time When raw time is compared to an external reference time source in order to adjust for clock drift, then the result is *adjusted raw time*. This adjustment doesn't happen sporadically but rather, the rate of advance of time is slowed down or sped up slightly until it approaches that of the external reference time source. Therefore, adjusted raw time is still monotonic. Like raw time, adjusted raw time is subjective with no specific meaning attached to its values. But how does one obtain the adjusted raw time? One way to do this to access an external time source using one of the networking timing protocols we discussed earlier. The approach, however, is thus susceptible to some of the security risks that underly these network timing protocols.

4.2.2 The Domain Name System (DNS)

We briefly review the aspects of DNS, address resolution and caching that are relevant to the DNS cache attacks that we present in Section 4.3.

Domain Name System DNS is one of the Internet's most critical components. DNS is a distributed database that provides mappings between domain names and IP addresses and other resources in the form of *resource records* (RRs). A DNS name server (NS) stores the DNS RRs for a domain and responds with answers to queries against its database. An *authoritative NS* is responsible for mapping domain names for a specific domain (*e.g.,* `example.com`) to Internet resources, by serving the relevant DNS RRs (A, CNAME, PTR, *etc.*).

Caching To resolve an address (Mockapetris, 1987a), a *stub resolver* typically sends DNS query from an end-user system to a recursive resolver. If the requested record is not found in the recursive resolver's cache (*i.e.,* the query 'cannot be resolved locally') then the recursive resolver recursively queries the authoritative NSs in order to resolve the name. The recursive resolver caches the final response, and any additional RRs it learned as part of the resolution process, and sends the final response to the stub resolver. The stub resolver may also cache the final response.

4.2.3 Time in DNS

Time is an important component of DNS RRs. Each RR contains a time interval, known as the Time to Live (TTL), which is assigned by the administrator of the DNS zone in the zone file (*aka.,* "master file" in (Mockapetris, 1987a)). The DNS specifications require the TTL to be a relative time. However, as we shall now see, these TTLs are converted to absolute time when they are used for DNS caching. Later, in Section 4.3, we show that this conversion to absolute times creates network-time-related vulnerabilities.

According to RFC 1035 (one of the earliest DNS RFCs), TTL specifies the time interval (*i.e.,* the relative time) that an RR may be cached before the source of the information should be consulted again (Mockapetris, 1987b). RFC 1035 recommends that all caching resolvers obey TTL values for caching, but RFC 2181 (Elz and Bush,

1997) says that caches can upper-bound the TTL of any RR, and treat any TTLs larger than the upper bound as if they were equal to the upper bound. In other words, the TTL specifies a maximum time to live, *not* a mandatory time to live. RFC 1035 also advises the zone file manager about suitable TTL values for different RRs in different situations.[3]

Upon receiving a DNS RR, a caching resolver is supposed to cache the RR for the time interval specified by its TTL. While the TTL is a *relative* value, the RFCs do *not* clearly specify how the cache should determine that the TTL has elapsed. However, there is some evidence that the RFCs *assume* that the cache converts the TTL to an absolute time.

The first bit of evidence comes when RFC 1035 says "When the RR has an absolute time, it is part of a cache", which suggests that the RFC assumes that the cache will implement the TTL as an absolute time.

The second bit of evidence comes from RFC 1035's recommendations for validating the freshness of an RR that is retrieved from the cache in order to answer a query. Specifically, RFC 1035 recommends timestamping the query for an RR using the current system time, and comparing that timestamp with the TTL of the RR in the cache. Comparing the system time (an absolute time) with the TTL time implicitly implies that the cache is storing the TTL as an absolute time.[4]

We checked popular caching resolver implementations ((Unbound, 2018), (Bind, 2018), (Powerdns, 2018), (Dnsmasq, 2018), (knot, 2018) (before v1.5.1)) to see how

[3]For instance, RFC 1035 says that TTL values on the order of days or weeks boost Internet performance and suggests a TTL value of zero for certain records such as SOA RR that should not be cached. One may also choose to have lower TTL values for extremely volatile data or if a change in the RR is anticipated to minimize inconsistency and then later revert back to the longer TTL.

[4]Specifically, RFC 1035 says: The "timestamp indicat[es] the time the request [for an RR] began. The timestamp is used to decide whether RRs in the database can be used or are out of date. This timestamp uses the absolute time format previously discussed for RR storage in zones and caches. ... When the RR has an absolute time, it is part of a cache, and the TTL of the RR is compared against the timestamp for the start of the request."

their caches were implementing time. We found that these implementations all mark the end of validity of the cached object by translating the relative time values in the TTL into absolute time values by adding an offset equal to the TTL to the current system time.

DNS resolver implementations do not come with a predefined mechanism for getting absolute time. So the best that they can do is to rely on system time (which represents some form of absolute time) from underlying OS to get these absolute time value. For instance, the POSIX function to get the system time is `gettimeofday()`, which gives the number of seconds and microseconds since the epoch `1970-01-01 00:00:00 +0000 (UTC)`. It therefore follows that DNS resolver implementations take the correctness and security of system time for granted. This creates a security risk, because the system time is often set by network timing protocols (*e.g.,* NTP), which are vulnerable to attack (see Section 4.2.1.)

Cache security and correctness. Finally, we note that the risks due to the cache's reliance on absolute time (or system time) are not mentioned in the DNS RFCs. RFC 1035 only minimally discusses security risks to the cache. The closest that RFCs get to this is: (a) RFC 1035 suggests implementing the cache as separate data structure so that it can be easily discarded without disturbing zone data (especially since the cache is vulnerable to corruption when a system reboots, and because it can also become full of expired RRs) and (b) RFC 3833 (Atkins and Austein, 2004), RFC 5452 (Hubert and von Mook, 2009) and RFC 7873 (III and Andrews, 2016) discuss hardening caches to DNS spoofing and cache poisoning attacks.

4.3 Using Time to Attack the DNS Cache

We now investigate a largely overlooked and important threat in DNS: the impact of security vulnerabilities introduced because resolver's caches are dependent on absolute

time, which is obtained from the system time, which is often obtained via network timing protocols (*e.g.,* NTP), which are vulnerable to attacks. In this section, we present two attacks that exploit vulnerabilities in networking timing protocols: a *Cache Expiration Attack* and a *Cache Sticking Attack*. We also discuss how each of these attacks can harm DNS and other protocols and applications that rely on DNS.

4.3.1 Cache Expiration Attack

If the system time is shifted forward on a caching DNS resolver, then the RRs in the cache would expire sooner than intended. This is tantamount to flushing attacks on DNS cache and may cause a denial of DNS services to a client.

This attack is possible when the system time is updated from an external source that is attackable, *e.g.,* NTP (Malhotra et al., 2016), (Malhotra and Goldberg, 2016), (Malhotra et al., 2017) and SNTP (Selvi, 2014). The attack follows because DNS cache implementations use the system time to determine TTLs. Meanwhile, the DNS protocol gives TTL as a relative time. If the cache had instead implemented TTLs as a relative time (*i.e.,* raw time or adjusted raw time, see Section 4.2.1) the security of system time would have no impact on the security of the cache.

We perform the attack in the laboratory setting. For this we set up our own validating, recursive, and caching DNS resolver Unbound v1.7.3 (Unbound, 2018) on an Arch Linux version 2018.08.01 machine. (This was the latest version of Unbound and Arch Linux at the time of our test). Our resolver uses NTP to update the system time; indeed, NTP is the most common method of time synchronization on GNU/Linux systems and on Arch Linux systems (Archlinux, 2018). We reboot the system just to allow for a clean attack demonstration; this same attack is possible even if the system is not rebooted. On system reboot, the resolver cache is empty. We then perform the following experiment:

1. We initialize the empty cache by inserting an `A` record as follows. Send an

`A` record query to the recursive resolver for the domain `ndss-symposium.org` using the standard `dig` query for DNS lookup. (The `dig` query bypasses the stub resolver cache lookup on our test machines.) Since the recursive resolver has an empty cache, it performs a recursive lookup for the domain. It first queries the root TLD, then .org authoritative NS and then ndss-symposium.org authoritative NS to get the final A record as the answer. In our experiment, the recorded time to complete the entire recursive address resolution process was 267 ms and TTL was 3600 sec $\approx$ 1 hour (per the"Query time" and "TTL" field respectively in the `dig` response message).

2. Next, we confirm that the `A` record has been cached by the resolver. To do this, we perform another query, within the TTL window of the `A` record, for the same domain.The record time to answer the query was just 4 ms, implying that the record is served from the resolver cache.

3. Next, we attack by manually changing the system time on the resolver to be past the TTL window of the cached `A` record by 1 hour 5 minutes. (There are several techniques to perform these attacks from off-path and on-path described in (Malhotra et al., 2016), (Malhotra et al., 2017), (Malhotra and Goldberg, 2016), (Selvi, 2014). We did not reproduce those attacks in our lab.)

4. Next, we perform the same query to the resolver. We observe now the query took 94 ms because it did not have to do the full recursion from the root. The root and .org were still cached (because they have longer TTLs than 1 hour 5 minutes.) The resolver had to refetch the `A` record for only `ndss-symposium.org` which indicates that the `A` record was flushed from the cache.

4.3.2 Implications of the Cache Expiration Attack

We consider the implications of cache expiration attack in three categories: (a) impact on DNS, (b) making other attacks easier, and (c) economic repercussions for outsourced domains.

Impact on DNS. Caching is critical to the DNS. *All* recursive resolvers are caching, while many stub resolvers (depending on the underlying OS) support caching. (For instance, Windows (Klein, 2018), Linux dnsmasq (dnsmasq, 2018) and Linux systemd (systemd, 2018) do support caching in stub resolvers, but the libc stub resolver does not.) Our attack harms caching performance and availability.

Performance A cache fetch from the resolver is much faster than a full recursive query to one (or many) external NS(es). This is also indicated in our attack demonstration above which took 4 ms to respond from the cache as opposed to 267 ms for full recursive response. Research suggests that resolvers typically have a 70-90% cache hit rate (Jung et al., 2001), (DeGroote, 2013), so by flushing the cache using our cache expiration attack, we can harm performance significantly. Each time this attack is launched, flushing the cache, it harms the performance for the first query made for a given RR. Nevertheless, we believe this performance hit is significant. As evidence of this, (Kareem, 2017) observed that DNS resolver operators sometimes serve RRs from the cache for 2-3 days longer than is allowed by their TTLs. In other words, the the performance hit we have demonstrated here is important enough to DNS operators that they sometimes are willing to disregard the requirements of the DNS specifications.

Availability If the cached RRs expire too soon and the local caching resolver has to go out and query the name servers for every other query it receives, the reliability level of the DNS becomes the limiting factor in availability of service. RFC 1034 (Mockapetris, 1987a) says that even though the Internet is built for resiliency

and redundancy, nameservers can be down and/or communication link to the root server broken due to network disruptions or other reasons. In such cases, the cache of recursive resolver decides the availability of resolving services.

This is a real threat, as we have seen many instance when DNS services go down but users are unaffected because of the presence of the cache. Consider, for instance large-scale DoS attacks like as DNS flooding, which are symmetrical DoS attacks that attempt to exhaust server-side assets (e.g., memory or CPU) with a flood of UDP requests, generated by malicious scripts running on several compromised botnet machines. In one incident (rootops, 2015), some root DNS servers received high query rates that caused network connections to saturate, denying service to valid, normal queries. As another example, series of DDoS attacks on DNS provider Dyn in 2016 ("2016 Dyn cyberattack", 2016) affected many major services. (News headlines went so far as to say that it "broke the Internet" (Gordon, 2016), (SiteUptime, 2016), (Steinberg, 2016)). When Dyn's targeted, authoritative DNS servers became unavailable, the attack traveled across the world at TTL speed. That is, as soon as a recursive resolver attempted to refresh its cache and discovered that Dyn wasn't available, it had no supply of IP addresses to provide. Since most of the records in question relied on short TTLs, the impact was almost immediate. Meanwhile, longer TTLs in DNS cache buys time for remedial action to be taken against the adversary. However, with our cache expiration attack if the cached responses expire, then the service is no longer available.[5]

Other Attacks made easier. We can use the cache expiration attack to make other DNS attacks easier to accomplish.

DNS cache poisoning made easier With DNS cache poisoning, an attacker at-

[5] As a solution, there is a new draft IETF specification (Lawrence and Kumari, 2017) that proposes to serve expired data from the cache to maintain availability and draft. Also (Pappas et al., 2012) suggests keeping longer TTL values to improve DNS service availability during prolonged outages.

tempts to insert a fake DNS record into the cache. If the querying resolver accepts the fake record and saves it, the cache is poisoned and subsequent requests for that record are answered as this fake record within its TTL window. The attacker may poison the cache with fake A or AAAA, or CNAME, or NS RR that cause traffic redirects to a server controlled by attacker. Since almost all communications on the Internet require a DNS lookup, any application that is served from the DNS cache becomes vulnerable.

Importantly, cache poisoning attacks can be difficult because they require the victim resolver to send a query to an external NS. However, if the relevant RR is already present in the victim resolver's cache, then the query to the external NS will never be sent, and the cache poisoning attack will not succeed. (Indeed, RFC 5452 explains how longer TTLs for cached responses make DNS cache poisoning harder.) Meanwhile, our cache expiration attack flushes RRs out of the cache, making it easier to launch a cache poisoning attack.

Fast fluxing made easier Fast Flux (Honeynet, 2018), (Holz et al., 2008) is a DNS technique used by botnet networks to hide various malicious activities, behind a dynamic network of compromised machines acting as proxies. The basic technique behind fast fluxing is to have multiple IP addresses associated with a single domain name (controlled by the attacker) which are swapped at high frequency, using short TTL values. However, fast fluxing is commonly thwarted by DNS resolvers that defensively reject RRs with very short TTLs. However, with our cache expiration attack, one can effectively shorten TTLs short, thus thwarting this defense mechanism and aiding fast fluxing. While our attack works only on individual caching resolvers (rather than authoritative NSes), it's important to remember that some resolvers serve as core Internet infrastructure that are used by a large number of users (*e.g.,* 8.8.8.8, 9.9.9.9, 1.1.1.1). Moreover, fast fluxing via cache expiration attack is more

difficult to detect than very short TTLs.

4.3.3 Cache Sticking Attacks

If system time is shifted backwards on DNS resolver, the validity period of cached RRs may increases. This makes them stick in the cache for longer than their intended TTL. This attack is identical to the Cache Expiration Attack we just presented, except that now, the attacker maliciously shifts time backwards, rather than forwards. That said, the implications of this attack are different.

4.3.4 Implications of the Cache Sticking Attack

Cache Poisoning lasts longer If the attacker is able to poison the cache at a victim resolver (by inserting a fake RR into the cache), then by sticking the cache he can keep the cached poisoned for a long time. This approach circumvents many existing defenses that try to limit the impact of cache poisoning by limiting the TTL of records that live in the cache. For instance, by default the Unbound resolver will requery for cached responses *every* 24 hours to alleviate the damage if the cache is poisoned. (This way a poisoned record will necessarily be requeried in 24 hours, forcing the attacker to relaunch the attack every 24 hours.) Meanwhile, our cache sticking attack will prevent this requerying, since the victim resolver will not realize that 24 hours have passed.

Domain Propagation Delay The TTL of a RR determines the rate at which changes to a DNS RR propagate through the DNS ecosystem. This is why RFC 1035 suggests that when a change to an RR is anticipated, the domain owners should reduce the TTL values of that RR prior to the change. Meanwhile, our cache sticking attack thwarts this approach, by making even those RRs with a short TTL live longer in the cache. While this attack targets resolvers (rather than authoritative NSes), it can still have a big impact if those resolvers serve as infrastructure (*e.g.,* 8.8.8.8 or

9.9.9.9 or 1.1.1.1). We now discuss several ways that this cache sticking attack can harm DNS.

1) Failover. Content Delivery Networks (CDNs) often use short TTLs on DNS records to tackle failovers. Failover is defined to be the process of moving an service from one IP to another, in the event that access to that IP fails. This failover process should happen as quickly as possible. A low TTL for the `A` record bindings can reduce the failover latency due to DNS caching and update the system more quickly, making the failover services more effective (Jung et al., 2001), (Singh and Schulzrinne, 2007). However, with our cache sticking attack, this approach of adding robustness to the system can be rendered ineffective by sticking old RRs in the cache and delaying the propagation of new RRs.

2) Load Balancing and Quality of Service (QoS): Various CDNs (Cloudflare, 2018) (Akamai, 2018) rely on DNS-based server selection to balance load. The idea is that the CDN's authoritative NSes decide which RRs to serve in response to a query (*e.g.,* for `www.example.com`) based on the location of querying resolvers. (For instance, a resolver in Italy may be directed to a webserver for `www.example.com` at IP 1.2.3.4 while a resolver in China might be directed to a webserver for `www.example.com` at IP 9.8.7.6.) By serving different RRs to different resolvers (that all make the same query), the CDN can balance load across multiple webservers. Since it is a dynamic server selection scheme that must adapt to dynamically changing loads, CDNs may want to frequently change the RRs they serve to a given resolver. This requires TTLs on RRs to very short (on the order of a few minutes (Pan et al., 2003)). However, our cache sticking attack sticks a user to a particular RR, which may be outdated or saturated with load. This affects the quality of service achieved by the load balancing scheme.

3) Suppose a domain name is sold or transferred, orr that one wants to change the

DNS host for a domain. Then, the RRs related to this domain need to be changed. Short TTLs allow this change to quickly propagate through the DNS ecosystem. Meanwhile, with our cache sticking attack, an old RR may be stuck in the victims cache, directing the users to an old IP which may now be malicious or may have old data. (This idea is also similar to the IP-use-after-free attacks presented by (Borgolte et al., 2018).)

Negative Caching RFC 1034 (Mockapetris, 1987a) optionally provided a negative caching service to allow negative responses (NXDOMAIN) with TTLs to be distributed by NSes and cached by resolvers. RFC 2308 (Andrews, 1998) makes negative caching -the storage of knowledge that something does not exist - mandatory for the caching resolvers. This is important, as it not only reduces the response time of negative responses, but also helps reduce DNS traffic. However, RFC 2308 is sensitive to risk of a DoS attack that propagates via a negative response (NXDOMAIN) that has a very high TTL; as a defense, the RFC suggests a sanity check to make sure that the TTL on negative responses is not too high. Meanwhile, our cache sticking attack would thwart any such defense, by sticking even those NXDOMAIN responses with short TTLs in the cache for a very long time.

Blacklisting DNS has become the de-facto standard for the creation and distribution of blacklists of IP addresses/domains associated with spam and other anti-social behavior on the Internet. RFC 5782 says that a blacklist of IPs that sends a spam should have short TTLs as these IPs tend to change frequently (might change every few minutes). However, with the cache sticking attack, one can effectively make TTLs longer, potentially blacklisting address long after they have recovered from a compromise by the spammer.

4.4 Measuring the attack surface

We use measurements to find the number of DNS resolver IPs in the wild that are vulnerable to our *cache sticking* and *cache expiration* attacks.

We perform our measurement study on two kinds of DNS resolvers (1) *Open resolvers* - ones that are publicly accessible, and willing to resolve recursive queries for *anyone* on the Internet (2) *Private resolvers* - ones that are meant to provide resolver service to hosts in their respective network only. We test private resolvers only in those networks that have RIPE Atlas (ripe, 2018) probes.[6]

4.4.1 Pivoting from NTP time-shifting attacks

Our attacks rely on shifting time on the resolvers. Resolvers that use NTP to update their system time are vulnerable to off- and on-path time-shifting attacks (Malhotra et al., 2016), (Malhotra and Goldberg, 2016) and (Malhotra et al., 2017). Thus we measure the attack surface of our DNS cache sticking and DNS cache expiration attacks by assuming that the attacker preforms the attack via NTP.

MiTM attacks via NTP. All NTP clients are vulnerable to Man-in-the-Middle (MiTM) time-shifting attacks. This follows because NTP packets are never encrypted and are typically not authenticated by a message authentication code (MAC) (Malhotra et al., 2017). Instead, an NTP client decides whether to accept packets from a timeserver by checking a nonce called the *origin timestamp* on the timeserver's response packet; the origin timestamp on the timeserver's response packet (*aka,* an NTP *mode 4* packet) is required to match the *transmit timestamp* on the client's query packet (*aka,* an NTP *mode 3* packet). The transmit timestamp on the client query can trivially be read by a MiTM, making NTP vulnerable to MiTM attackers that spoof bogus server response packets in order to shift time on the client (Malhotra

[6]Since these networks are maintained by individuals or enterprises for internal use, they are generally not accessible from outside the network.

et al., 2016), (Malhotra et al., 2017). If a resolver is vulnerable to NTP MiTM attacks, the MiTM can execute our cache expiration and cache sticking attacks on DNS. As we will soon see, we found that 33% of measured private resolvers and 5.37% of measured open resolvers speak NTP and are thus vulnerable to these MiTM attacks. Below we discuss how we find and interpret these results.

Off-path attacks via NTP. Next we consider whether NTP can be used to launch our cache expiration and cache sticking attacks from off-path. One key reason that NTP is vulnerable to off-path attacks is because it operates over UDP. Nevertheless, clients that simply speak NTP are not necessarily vulnerable to off-path time-shifting attacks (although several generic off-path time-shifting attacks have been found and patched (Malhotra et al., 2016), (Malhotra et al., 2017)). This follows because an off-path attacker cannot read the transmit timestamp nonce on an NTP client query, and therefore cannot correctly inject a bogus server response with a correct origin timestamp. To attack from off-path, the attacker somehow has to learn the origin timestamp nonce.

That said, there is a known attack (Malhotra et al., 2017, Section V), called the "Leaky Origin Timestamp" that causes the client to reveal his nonce. This attack is possible on clients that answer unsolicited NTP *mode 6* control queries from arbitrary IPs with information that reveals the origin timestamp nonce. The specific NTP control queries that should be answered to launch this attack are either (`rv assocID org` or `rv assocID rec`). While patches for this attack exist (namely: stop answering NTP control queries from unknown IPs), many machines remain vulnerable even as of August 2018. Thus, we will decide that a DNS resolver is vulnerable to our DNS cache expiration and cache sticking attack if they respond to either one of the above NTP control queries. Surprisingly we find that 18.28% of the open resolvers that replied to NTP mode 3 queries and 6% of private resolvers that replied to NTP mode

3 queries leaked at least one of the `org` or `rec` nonces.

Finally, we note that there is another off-path time-shifting attack called the "Zero-Origin Timestamp Attack" (Malhotra et al., 2017, Section IV.A). While this attack was patched over two years ago, not all machines have deployed the patch. This attack is especially powerful because it does not require the victim NTP client to answer unsolicited mode 6 control queries from arbitrary IPs. That said, to *measure* the attack surface for this zero-origin time attack, we need to use NTP mode 6 control queries (Malhotra et al., 2017). When performing this measurement, we found that of the resolvers that answered unsolicited mode 6 control queries, we found that 70.17% of open resolvers and 69.14% of private resolvers were vulnerable to the zero-origin timestamp attack.

The bottom line is that many resolvers in our dataset are vulnerable to known attacks on NTP.

4.4.2 Private resolver measurements.

To find out the number of DNS resolvers that are vulnerable to attacks that pivot from NTP, we use the RIPE Atlas probes. At the time of our experiment (July 29 2018 - August 2 2018), a total of 10,338 Atlas probes were active. Each probe gets its list of DNS resolvers by either manual configuration or by default by DHCP in their respective networks. We start by obtaining the list of IPs for DNS resolvers on each probe from the publicly available data from another - long running - DNS measurement study (ripe NCC, 2018) on RIPE network. This study builds the resolver IP list by sending a DNS `TXT` query for `o-o.myaddr.l.google.com` from each probe using the locally configured resolvers. The IP addresses of the locally configured resolvers are part of the measurement results. By using the results of this query measurement, we make sure to select only working local resolvers.

In the following steps we discuss our measurement methodology and the challenges

that we face.

1) MiTM attacks. First we send NTP mode 3 client queries to these privates resolvers. From the ones that respond to these queries with a mode 4 server response, we can determine the number of resolvers that use NTP to update their *system time*. Importantly, this only gives us a lower bound on the number of resolvers that are NTP clients, since a security-aware NTP client will only *send* NTP mode 3 client queries but not *respond* to them (Malhotra et al., 2016). NTP clients that respond to NTP mode 3 queries are (often unwittingly) acting as NTP timeservers. We emphasize that there is a significant probability that there are other resolvers that are NTP clients but are configured to ignore mode 3 queries. While an MiTM can launch our DNS cache attacks to any resolver that speaks NTP (even those who are not acting as timeservers), we know of no other technique to determine if a remote machine uses NTP to set its system time. However even this measurement comes with a challenge.

Hurdle 1: Atlas probes allow NTP mode 3 queries *only* to public IP addresses.[7]. In order to filter out the resolvers that have private address, we discard the resolvers that have IPs in the private address space as described in RFC1918 (Rekhter et al., 1996). We also discard well-known open DNS resolver addresses.

Stats. We then use the remaining IP addresses as target for mode 3 query. After filtering we obtained 4,703 unique resolvers with public IP address. We then send RFC5905-compliant mode 3 NTP queries to these resolvers from the Atlas probes that had them configured. (Note: NTP mode 3 queries do *not* modify the internal state of the queried systems.) Of the 4,703 queried resolvers, a staggering 1,535 (32.64%) resolvers answered with a mode 4 response to our mode 3 NTP queries.

Discussion The fact that 33% of the scanned private resolvers are acting as NTP timeservers (by responding to NTP mode 3 queries) looks pretty odd. But if we think

[7]Note that this restriction is only for NTP queries. One can send DNS queries to both public and private IP addresses.

about it, we expect a properly-configured DNS resolver to speak NTP, since accurate time is a good operational practice, and a necessity for DNSSEC validation too.

Why then, are so many resolvers acting as NTP timeservers? This is likely because most DNS operators do not spend time thinking through the intricacies of NTP. It turns out that, by default, the ntpd daemon makes every NTP-speaking system run as an NTP server (Malhotra et al., 2016); this functionality needs to be turned off manually by the administrator. Unfortunately, most DNS resolver administrators are unlikely to be aware of this default. NTP-speaking systems that act as timeservers are also more vulnerable to time-shifting and DoS attacks, when compared to NTP-speaking systems that have turned off their timeserver functionality (Malhotra et al., 2016).[8] Also, most DNS operators are probably unaware (or have not thought deeply about the fact) that NTP is trivially vulnerable to time-shifting attacks by an MiTM, because NTP packets are not cryptographically authenticated.

1) Off-path attacks. Which of these 1,535 NTP-speaking DNS resolvers are vulnerable to off-path time-shifting attacks? To answer this question, we use NTP mode 6 control queries to see how many of those leak their `org` and `rec` nonces. We run a script that sends the following mode 3 queries in order.

```
ntpq -c "assocID"
ntpq -c "rv assocID org"
ntpq -c "rv assocID rec"
```

As described earlier, systems that reply to the above queries by revealing either `org` or `rec` nonce are vulnerable to the off-path "Leaky Origin Timestamp" time-shifting attack on NTP (Malhotra et al., 2017). This brings us to our next challenge.

Hurdle 2: Atlas probes *do not* allow mode 6 NTP queries to be sent. This means

[8]This follows because an NTP mode 4 server response reveals the IP address of the time source used by the responding timeserver, which is useful for launching off-path time-shifting attacks (Malhotra et al., 2016).

that we can not query these resolvers from inside their network using Atlas probes. We found two ways that can, to some extent, solve this problem.

One solution is to querying these resolvers from outside the network in which they reside using NLnet Labs network. This approach is very limited since the resolvers are inside a private network, there is a low chance that they can be queried from outside. We queried the 1,535 resolvers. 68 resolvers leaked at least one of the `org` or `rec` nonces. Of these, 49 resolvers also leaked information indicating that they are vulnerable to the NTP Zero-Origin Timestamp attack.

An additional solution is to use the NLNOG RING (Snijders, 2018) nodes. 377 (24.56%) resolvers have a common network with at least one NLNOG ring node. Of these 42 (11.15%) resolvers leaked at least `org` or `rec`, and 38 of the responding resolvers also revealed that they are vulnerable to the Zero-Origin Timestamp attack on NTP.[9]

Putting these two approaches together, we found a total of 98 resolvers that revealed at least one of the `org` or `rec` NTP nonces (and thus are vulnerable to the off-path"Leaky Origin Timestamp" attack). Of these 65 resolvers were also vulnerable to the off-path "Zero Origin Timestamp" attack on NTP. Both of these NTP attacks should have been patched years ago. Nevertheless, these resolvers remain vulnerable, which by extension makes them vulnerable to our DNS cache attacks from off-path.

4.4.3 Open resolver measurements.

To measure the number of DNS resolver IPs in the wild, we obtain data from the Open Resolver Project (Mauch, 2018) that runs weekly scans of IPv4 address space to determine the IPs that respond to DNS queries. obtained a list of 23,137,317 IPs that responded to these queries during the week of April 15, 2018. We then start our

[9]NLNOG RING nodes do not have inbuilt support for mode 6 NTP queries. However, NLNOG RING allows for customized queries, so we copied NTP mode 6 queries to these nodes.

Resolver type	Total IPs	replied to `mode 3`	leaked `org` OR `rec`	leaked `org` 0
Open	342,131	18,385 (5.37%)	3,434 (18.68%)	2,416 (70.17%)
Private	4,703	1,535 (32.64%)	98 (6.39%)	65 (69.14%)

Table 4.1: Resolver IPs vulnerable to NTP time-shifting and our DNS cache expiration and cache sticking attacks. Column 3- All IPs vulnerable to MiTM, Column 4 - All IPs vulnerable to off-path attacks, column 5 - All IPs vulnerable to off-path Zero Origin Timestamp attack.

scan on DATE July 28, 2018 as follows:

1) We send a DNS `A` query for `www.example.com` to this list of 23,137,317 IPs. 1,458,194 IPs still replied to the DNS `A` query. Of the ones that replied, we found that 342,131 (23.46%) are open resolvers (those that replied with an answer) and 1,116,063 (76.54%) were *non-answering*, returning either REFUSED or some other DNS error code, indicating that they resolvers, but not open resolvers.[10]

2) Next, we send NTP mode 3 query to 342,131 answering open resolver IPs. Of these, 18,385 (5.37%) replied back with a mode 4 response packet. And 87,141 (7.80%) of 1,116,063 non-answering IPs replied back with a mode 4 response. *All* of these resolvers are vulnerable to MiTM NTP attacks and hence our DNS cache expiration and cache sticking attacks by an MiTM.

3) We then send NTP mode 6 control queries to 18,385 answering open resolver IPs and found that 3,451 (18.77%) IPs leaked at least one of the `org` or `rec` nonces. Moreover, a total of 416 (70.17%) resolvers indicated that they are vulnerable to the NTP "Zero-Origin Timestamp" attack.

Discussion 5.37% of scanned open resolvers answered NTP mode 3 queries. We don't expect open resolvers (with the exception of infrastructure open resolvers like 8.8.8.8 or 1.1.1.1) to be properly configured; the fact that they are open resolvers is sufficient evidence of misconfiguration. Why are open resolvers (5%) less vulnerable

[10]These non-answering resolvers are properly configured private resolvers with access control lists (ACLs). They may have been open resolvers at those IPs when openresolver project collected the data, but became non-open resolvers at the time of our scan.

than private resolvers (33%) to NTP MiTM attacks? Again this might look odd, but this may be because nobody cares to configure NTP on open resolvers in the first place, which could be viewed as a bad DNS operational practice. Also, a staggering 18.68% open resolvers acting as NTP timeservers (as compared to only 6% private resolvers) were found to be vulnerable to off-path time-shifting attacks. This again supports our argument of misconfiguration, except that now it is a bad configuration for NTP.

4.4.4 Takeaways and Ethical Measurements

We observe that more private resolvers (as compared to open resolvers) have NTP configured (33%) (good for DNS(SEC) operational practice), but fewer of them are vulnerable to off-path time-shifting attack (6%) (good NTP configuration). Fewer open resolvers are NTP configured (5%) (bad DNS(SEC) operational practice), but more of them are vulnerable to off-path time-shifting attacks (19%) (bad NTP configuration). In other words, if the DNS resolver is sloppily configured, then NTP is sloppily configured as well. Therefore we ask: Is the risk of sloppily configuring NTP bigger if one configures NTP at all? Our results suggest that this is indeed the case.

Therefore, DNS resolver operators that decide to turn on NTP should also be careful with their NTP configurations. DNS resolvers should not be operating at NTP timeservers, because this increases the surface for NTP attacks (Malhotra et al., 2016). DNS resolvers should also regularly update their NTP software, to avoid situations where NTP vulnerabilities remain exploitable two years after they should have been patched.

Ethical measurements. We acknowledge that a far more 'informative' measurement would have been to directly measure our DNS cache attacks in the wild by launching NTP time-shifting attacks in the wild, and then checking if the DNS cache has been flushed or stuck. Naturally, however, we did not do this, because it would be

unethical to attack live resolvers. This is especially important since our DNS cache attacks indiscriminately flush or stick *all RRs* in the cache, so we could not have a done a very targeted "test attack" on our own RRs that we inject into the cache of the measured resolver (as was done *e.g.,* in (Klein et al., 2017)). Instead, we focused on using NTP side channels for measurements. None of the side channels modify the internal state of the NTP daemon (see also (Malhotra et al., 2016), (Malhotra et al., 2017)), and thus also do not modify the state of the DNS cache.

4.5 Recommendations

We believe that setting TTL as *relative time* value is a feature of the DNS protocol. To deal with the attacks described in Section 4.3, there is no fundamental change required to the DNS protocol. The only thing that needs to be dealt with is the way *relative time* values (TTL) are implemented in DNS cache. Since TTL represents a duration of time, we just need the value that gives the difference in time between two points.

We recommend to get this value from a monotonic source of time that is not subject to a) manual changes and b) adjustments by vulnerable network timing protocols. Instead, one can use raw time or adjusted raw time,[11]as discussed in Section 4.2, depending on their availability on different OS. If both types are available, the choice of time value to be used is application-specific. For applications that can tolerate a certain amount of clock drift or need to keep track of extremely small intervals of time (say on the order of few seconds), then raw time should be used. However, if that is an issue, then one has no choice but to fall back to adjusted raw time.

[11]Note that adjusted raw time is subject to adjustments by timing protocols. However, 1) it can not be set manually, 2) it does not make large jumps, and 3) the rate of passage of time is just slowed down or sped up which means that it will take way longer to shift time. Adjusted raw time is better than system time but worse than raw time; see Section 4.2.

4.5.1 Implementing our Recommendations

Unbound Unbound is a validating, recursive, and caching DNS resolver product from NLnet Labs(Unbound, 2018). It is distributed free of charge in open source form under the BSD license.

Unbound's internal operation is organized around asynchronous event notification. Different serve-threads wait for events to happen (*i.e.,* network sockets to become readable, writable, closed or to fail) to take appropriate action. Events are registered with a so called "event-base" (or loop) together with a timeout value. Fired time-outs are also events upon which (different) appropriate actions are taken (*i.e.,* retry scheduling of queries at different authoritatives *etc.*).

Unbound serve-threads read and record system time, every time events become available (or timed out). (This is similar to the recommendation in RFC1035 to timestamp every request/query for an RR, that we discussed in Section 4.2.3.) The timestamp is subsequently used with Unbound's operations dealing with the fired events: to set expiration time for newly cached RRsets, to check the validity of already cached RRsets, to limit the rate at which requests are send to authoritative servers, and to check validity of DNSSEC signatures. The function which is used to read and record system time is `gettimeofday()`.

We have a preliminary implementation of our recommendations for Unbound, and the implementation is publically available on GitHub. [We omit the link to preserve anonymity.] At the time of writing, Unbound has been altered as follows. Instead of one, two timestamps are recorded when events are available, one absolute time value from `gettimeofday()` which is used for DNSSEC validation only, and one relative time value from `clock_gettime()` — currently with the `clock_id` parameter set to `CLOCK_MONOTONIC_RAW` (the unadjusted raw time on Linux systems) — which is used for all other timing related functions, such as dealing with RRset's TTL values

and rate-limiting requests to authoritative servers. Several issues still remain to be resolved:

1. Those tests — that are part of the Unbound source tree — which are testing timing behavior of Unbound (*i.e.,* cache expiration *etc.*) are failing, because the scripts for those tests are based on absolute time values.

2. The current implementation uses the Linux specific `CLOCK_MONOTONIC_RAW` clock which is an extension on the POSIX standard("clockgettime", 2018), which defines `CLOCK_MONOTONIC` only and leaves whether or not the value is adjusted for drift, undefined. In general, portability of the implementation has to be evaluated.

3. Furthermore, Unbound has a built-in event-base system based on select. The registration and handling of timeout values with this event-base are still based on absolute time values from `gettimeofday()`. This has to be rewritten to use relative time values too.

NLnet Labs has committed to incorporate our changes into Unbound's main development branch once all issues have been resolved and the implementation has proven to be stable and not negatively affecting current deployments.

Knot-resolver After a preliminary disclosure of our work at IETF 100, the Knot resolver (knot, 2018) co-incidentally also changed how it used time for DNS caching. This was released as Knot version 1.5.1. Because Knot's cache is persistent, it needs a defined epoch to be meaningful between restarts and reboots. The cache still has expiration values in absolute time values, but timing discontinuities are detected by comparing progression of system time with monotonic time. In default configuration, the Knot clears the cache if time jumps more than 10 minutes into the past. This has two implications.

1) This opens the DNS cache to an alternate cache expiration/flushing attack. The attacker needs to shift time backwards by *only* 10 minutes and the cache will be flushed, which is equivalent to our cache expiration attack in which the attacker has to shift the time forwards to pass the TTL window of the cached response. In fact, this new attack could be easier, since it is independent of the TTL of a particular RR.

2) Cache sticking attacks, on the other hand, would now require more persistent work by the attacker. Remember, for sticking the RRs in the cache the attacker has to shift time backwards within the TTL window of the RR to stick the RR in the cache (Section 4.3.3). Meanwhile, knot has now imposed a restriction of 10 minutes. The attacker can not shift time backwards by more than 10 minutes; if she did, the cache will be flushed instead of getting stuck. As result, performing a cache sticking attack is harder, because the attacker has to keep attacking at least every 10 minutes for as long as he wants to stick the RRs in the cache.

We conjecture that the Knot could not fix this problem entirely because Knot uses a persistent cache. It is not possible to store relative time values (raw time or adjusted raw time) in a persistent cache.

4.6 DNSSEC and System Time.

DNSSEC described in RFCs (Arends et al., 2005a) (Arends et al., 2005b) and (Arends et al., 2005c) is a set of protocols that adds a layer of trust by providing *data origin authentication* and *data integrity* to DNS lookups and exchanges. For authentication, DNSSEC associates cryptographic digital signatures, called the Resource Record Signature (RRSIG) records, generated using the zone's private key with the DNS Resource Record Set (RRSet)[12]. By checking its associated RRSIG, one can verify that

[12]RRset is a group of records with same label, class and type, but with different data. Typically RRsets are signed as opposed to signing individual RRs.

a requested DNS RRset comes from its authoritative NS and wasn't altered en-route. A *security-aware* resolver understands DNSSEC and is capable of using DNSEC RR types to provide DNSSEC services. When a security-aware resolver learns the zone's public key, it can validate the RRSIG as described in (Kolkman and Gieben, 2006).

RRSIG RRs have defined the *inception time* and *expiration time* to establish a validity period for the signatures and the RRsets covered by the signature. These inception and expiration fields are specified as absolute time values. This is *unlike* DNS TTLs, that are expressed as relative time values. Thus, unlike the DNS on-the-wire protocol, the DNSSEC on-the-wire protocol uses the notion of absolute time.

How do implementations deal with timestamps?In a typical software implementation (Unbound, Bind, PowerDNS, DNSMasq, *etc.*), the two absolute time values on RRSIG RRs are compared against the current system time. The current system time of the resolver MUST be between these two time values; otherwise the response is discarded either as an expired or not-yet-valid record.

DNSSEC RRs expired/not-yet-valid attacks If the system time of the security-aware validating resolver is shifted forward such that it is past the expiration time on RRSIG, then we can effectively make RRSIG RRs expire. On the other side, if the time is shifted backwards such that it is before the inception time on the RRSIG, then we can make them "not-yet-valid". In such a case, most resolver implementations (including but not limited to Unbound) shows the SERVFAIL status to the client and cache this failed response for 60 seconds after five retries with the same NS. This DNSSEC DoS is tantamount to DNS DoS attack. (Note, there is no fallback to DNS in case of too many SERVFAILs.)

So how can our attacks make DNSSEC DoS (aka DNS DoS) easier when there are multiple NSes? First, we note that the Unbound resolver (Dai et al., 2016) (and other resolvers) keep some statistics about NSes, and will stop querying an NS that causes

too many SERVFAILs (or other failures) and start querying other NSes instead. Thus, a traditional DNSSEC DoS works only if *all* relevant NSes are causing SERVFAILs (or other failures). Meanwhile, our attacks directly DoS DNSSEC at the resolver by shifting time, without controlling *any* NSes, lowering the bar for the attack. Our attacks would invalidate all signed data from all correctly configured honest NSes, and leave the resolver with no NSes to fall back to.

4.6.1 Recommendations

The use of absolute time is inherent to security protocols. This raises the question: Why don't we use relative time values to define the validity of cryptographic objects just like we do for non-cryptographic records? There are two reasons we think that absolute time is needed for DNSSEC.

Argument 1. The authority determining and setting the validity period on the object can be different from the one delivering the object. For example, setting the TTL value on DNS records is an operational matter and is thus left to the operators of the DNS zone. Zone operators can change the TTL values on non-signed or non cryptographic records even when they don't own it. The content of the cryptographically signed records (RRSIG RRs) are, however, determined by the signer of the records. When the signer is not also the zone operator, signer has no way to determine when the records will be queried for, and thus has to depend on cryptographically signed absolute time values to limit the validity of the record stored in the zone[13].

Argument 2. In a replay attack, an attacker records the data and maliciously resends it later. However if the data has absolute time values on it, that limits its lifetime. This is not possible with relative time values.

Thus, it becomes imperative for the validating resolver to use system time to

[13]Note however that DNSSEC signatures do contain the original TTL of an RRset, restricting the maximum TTL value with which the operator may deliver the RRs.

validate the absolute time values on RRSIG RRs. The only way to ensure the security of such a system is to secure the way system time is updated. On the bright side, there is an ongoing effort at Internet Engineering Task Force (IETF) community to secure NTP. *e.g.,* (Franke et al., 2018) is a proposal to secure NTP's client/server mode using Public Key Infrastructure and another RFC 8573 (Malhotra and Goldberg, 2019) that deprecates MD5-based symmetric key authentication, which is considered to be too weak, and recommends the use of secure AES-CMAC as a replacement. DNSSEC resolver operators should be amoung the first to adopt these NTP security updates.

4.7 Related Work

Attacks. Several works study DNS spoofing and cache poisoning attacks and propose potential solutions (Kaminsky, 2008), (Herzberg and Shulman, 2012a), (Klein et al., 2017), (Herzberg and Shulman, 2012b), (Schomp et al., 2014), (Yuan et al., 2006). These attacks focus on *how* the DNS protocol can be exploited to poison DNS resolver caches. By contrast our attacks show how to pivot from vulnerabilities in absolute time (*e.g.,* in NTP), to attacks that flush/stick the DNS cache. We also discuss how our DNS cache attacks can lower the bar for classic cache poisoning attacks by a) making them easier to launch, b) making them stick in the cache for longer. Finally, we note that existing work tends to propose security fixes to DNS on-wire protocol, while our recommendations focus on the DNS resolver cache implementations.

DNS measurement. Many previous studies measure DNS resolvers. For instance, studies have analyzed resolver performance (Boulakhrif, 2015), (Jung et al., 2001), the effectiveness of resolver caching (Jung et al., 2001), (Bhatti and Atkinson, 2011), or the classification of open DNS resolvers and their non-legitimate responses. (Ager et al., 2010) measures local resolvers from 50 commercial ISPs for responsiveness and

correctness and compares them with third-party popular open DNS resolvers. (Yu et al., 2012) measure how current caching resolver implementations distribute queries among a set of authoritative NSs. Our measurements are different because we focus on NTP-related vulnerabilities at DNS resolvers.

NTP measurement. In (Malhotra et al., 2016), (Malhotra and Goldberg, 2016), (Malhotra et al., 2017), the authors scan the IPv4 address space to find systems that are vulnerable to NTP attacks. While our attack surface measurement uses similar NTP queries (as in (Malhotra et al., 2017)) our measurements are specifically focused on open and private DNS resolvers.

4.8 Conclusion

We showed that DNS resolvers' dependence on absolute time makes them vulnerable to DNS caching attacks and DNS(SEC) DoS attacks. We studied the attack surface using network measurements, focusing specifically on the Network Time Protocol (NTP) as a vector for remotely attacking time. Our network measurements indicate that the attack surface is large . From total 342,131 measured open resolvers, we identified 18,385 open DNS resolvers that use NTP to set the absolute time. From total 4703 measured private resolvers, we find 1,535 private DNS resolvers that use NTP to set the absolute time. All these NTP-speaking resolvers are vulnerable to attacks by a MiTM that pivot from NTP time-shifting into DNS cache sticking / cache expiration attacks and DNSSEC downgrade attacks. In fact, we identified 3,451 out of 18,385 measured open DNS resolvers and 98 out of 1,535 measured private DNS resolvers that are vulnerable to NTP time-shifting attacks that can be launched from off-path, by any remote machine on the Internet with the ability to spoof UDP packets. While these off-path NTP attacks should have been patched over two years ago, many resolvers in the wild remain vulnerable.

Our attacks indicate that DNS caches should lessen their reliance on absolute time. Indeed, the DNS on-wire protocol does not actually need absolute time; instead, DNS packets only use relative time. We therefore recommend that DNS caching resolvers use relative time rather than absolute time. Specifically, we suggest using the operating system's raw time, which is monotonically increasing and not subject to adjustment by external sources (and thus immune to the network attacks on time explored here). We are currently working with Unbound resolver to implement and roll out our recommendations; our implementation is publicly available on Github.

Chapter 5

The Security of NTP's Datagram Protocol

5.1 Introduction

Millions of hosts (Malhotra et al., 2016), (Minar, 1999), (Murta et al., 2006), (Mauch, 2015), (Czyz et al., 2014a) use the Network Time Protocol (NTP) (Mills et al., 2010) to synchronize their computer clocks to public Internet timeservers (using NTP's client/server mode), or to neighboring peers (using NTP's symmetric mode). Over the last few years, the security of NTP has come under new scrutiny. Along with significant attention paid to NTP's role in UDP amplification attacks (Czyz et al., 2014a), (Krämer et al., 2015), there is also a new focus on attacks on the NTP protocol itself, both in order to maliciously alter a target's time (*timeshifting attacks*) or to prevent a target from synchronizing its clock (*denial of service (DoS) attacks*) (Malhotra et al., 2016), (Stenn, 2016). These attacks matter because the correctness of time underpins many other basic protocols and services. For instance, cryptographic protocols use timestamps to prevent replay attacks and limit the use of stale or compromised cryptographic material (*e.g.,* TLS (Selvi, 2015), (Klein, 2013), HSTS (Selvi, 2014), DNSSEC, RPKI (Malhotra et al., 2016), bitcoin (Corbixgwelt, 2011), authentication protocols (Klein, 2013),(Malhotra et al., 2016)), while accurate time synchronization is a basic requirement for various distributed protocols.

5.1.1 Problems with the NTP specification.

We start by identifying three fundamental problems with the NTP specification in RFC 5905, and then exploit these problems in four different off-path attacks on *ntpd*, the "reference implementation" of NTP.

Problem 1: Lack of respect for basic protection measures. The first issue stems from a lack of respect for `TEST2`, the mechanism that NTP uses to prevent off-path attacks. Off-path attacks are essentially the weakest (and therefore the most scary) threat model that one could consider for a networking protocol. An *off-path attacker* cannot eavesdrop on the NTP traffic of their targets, but can *spoof* IP packets *i.e.,* send packets with a bogus source IP. This threat model captures 'remote attacks' launched by arbitrary IPs that do not occupy a privileged position on the communication path between the parties. (See Figure 5·2.)

NTP attempts to prevent off-path attacks much in the same way that TCP and UDP do: every client query includes a nonce, and this nonce is reflected back to the client in the server's response. The client then checks for matching nonces in the query and response, *i.e.,* "`TEST2`". Because an off-path attacker cannot see the nonce (because it cannot eavesdrop on traffic), it cannot spoof a valid server response. Despite the apparent simplicity of this mechanism, its specification in RFC 5905 is flawed and leads to several off-path attacks.

Problem 2: Same code for different modes. NTP operates in several different modes. Apart from the popular *client/server mode* (where the client synchronizes to a time server), NTP also has a *symmetric mode* (where neighboring peers take time from each other), and several other modes.RFC 5905 recommends that all of NTP's different modes be processed by the same codepath. However, we find that the security requirements of client/server mode and symmetric mode conflict with each other, and result in some of our off-path attacks.

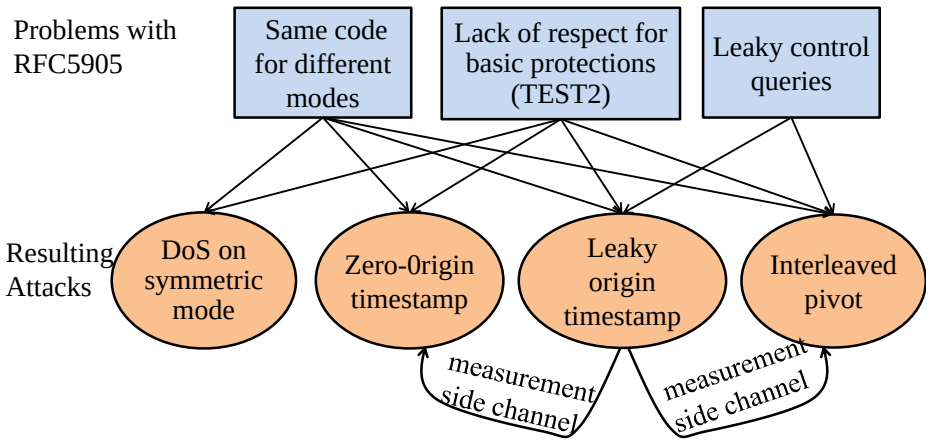

Figure 5·1: Chapter overview.

Problem 3: Leaky control queries. NTP's control-query interface is not specified in RFC 5905, but its specification does appear in the obsoleted RFC 1305 (Mills, 1992) from 1992 and a new IETF Internet draft (Mills and Haberman, 2016). We find that it can be exploited remotely to leak information about NTP's internal timing state variables. While the DDoS amplification potential of NTP's control query interface is well known (Czyz et al., 2014a), (Krämer et al., 2015), here we show that it is also a risk to the correctness of time.

We exploit these three problems to find working off-path attacks on ntpd (Section 5.3-5.4, Appendix A), and use IPv4 Internet scans to identify millions of IPs that are vulnerable to our attacks (Section 5.5). The first three attacks maliciously shift time on a client using NTP's client/server mode, and the fourth prevents time synchronization in symmetric mode.

Attack 1: Leaky Origin Timestamp Attack (Section 5.4). Our network scans find a staggering 3.8 million IPs that leak the nonce used in `TEST2` in response to control queries made from arbitrary IPs (CVE-2015-8139). An off-path attacker can maliciously shift time on a client by continuously querying for this nonce, and using it to spoof packets that pass `TEST2`.

Attack 2: Zero-0rigin Timestamp Attack (Section 5.3.3 and Appendix A.1). This

attack (CVE-2015-8138) follows from RFC 5905, and is among the strongest timeshifting attacks on NTP that has been identified thus far. The attacker bypasses `TEST2` by spoofing server response packets with their nonce set to zero. We use leaky NTP control queries as a side-channel to measure the prevalence of this attack. We find 1.3 million affected IPs. However, we expect that the true attack surface is even larger, since this attack itself does not require the control-query interface, works on clients operating in default mode, and has been part of ntpd for seven years (since ntpd v4.2.6, December 2009).

Attack 3: Interleaved-Pivot Attack (Section 5.4). Our third off-path timeshifting attack (CVE-2016-1548) exploits the fact that NTP's client/server mode shares the same codepath as NTP's interleaved mode. First, the attacker spoofs a single packet that tricks the target into thinking that he is in interleaved mode. The target then rejects all subsequent legitimate client/server mode packets. This is a DoS attack (Section 5.4, Appendix A.2).

We further leverage NTP's leaky control queries to convert this DoS attack to an off-path timeshifting attack. NTP's control-query interface also leaks the nonce used in the special version of `TEST2` used in interleaved mode. The attacker spoofs a sequence of interleaved-mode packets, with nonce value revealed by these queries, that maliciously shifts time on the client. Our scans find 1.3 million affected IPs.

Attack 4: Attacks on symmetric mode (Appendix B). We then present security analysis of NTP's symmetric mode, as specified in RFC 5905, and present off-path attacks that prevent time synchronization. We discuss why the security requirements of symmetric mode are at odds with that of client/server mode, and may have been the root cause of the zero-0rigin timestamp attack.

Disclosure. Our disclosure timeline is in Appendix E. Our research was done against ntpd v4.2.8p6, the latest version as of April 25, 2016. Since then, three versions have

been released: ntpd v4.2.8p7 (April 26, 2016), ntpd v4.2.8p8 (June 2, 2016), ntpd v4.2.8p9 (November 21, 2016). Most of our attacks have been patched in these releases. We provide recommendations for securing the client/server mode in Section 5.7 and symmetric mode in Appendix B.4.

5.1.2 Provably secure protocol design.

Our final contribution is to go beyond attacks and patches, and identify a more robust security solution (Section 5.6) We propose a new backwards-compatible protocol for client/server mode that preserves the semantics of the timestamps in NTP packets (Figures 5·6, 5·7). We then leverage ideas from the universal composability framework (Canetti, 2001) to develop a cryptographic model for attacks on NTP's datagram protocol. We use this model to prove (Section 5.6.3,5.6.4) that our protocol *correctly synchronizes time* in the face of both (1) *off-path attackers* when NTP is unauthenticated and (2) *on-path attackers* when NTP packets are authenticated with a MAC. We also use our model to prove similar results about a different protocol that is used by chronyd (chronyd, 2015) and openntpd (openNTPD, 2012) (two alternate implementations of NTP). The chronyd/openntpd protocol is secure, but unlike our protocol, does not preserve the semantics of packet timestamps.

Our cryptographic model models both on-path attackers and off-path attackers. An on-path attacker can eavesdrop, inject, spoof, and replay packets, but *cannot* drop, delay, or tamper with legitimate traffic. An on-path attacker eavesdrops on a copy of the target's traffic, so it need not disrupt live network traffic, or even operate at line rate. For this reason, on-path attacks are commonly seen in the wild, disrupting TCP (Weaver et al., 2009), DNS (Duan et al., 2012), BitTorrent (Weaver et al., 2009), or censoring web content (Clayton et al., 2006). Meanwhile, we cannot prove that NTP provides correct time synchronization in the face of the traditional Man-in-The-Middle (MiTM) attacks (*aka.* 'in-path attacks') because an MiTM can always

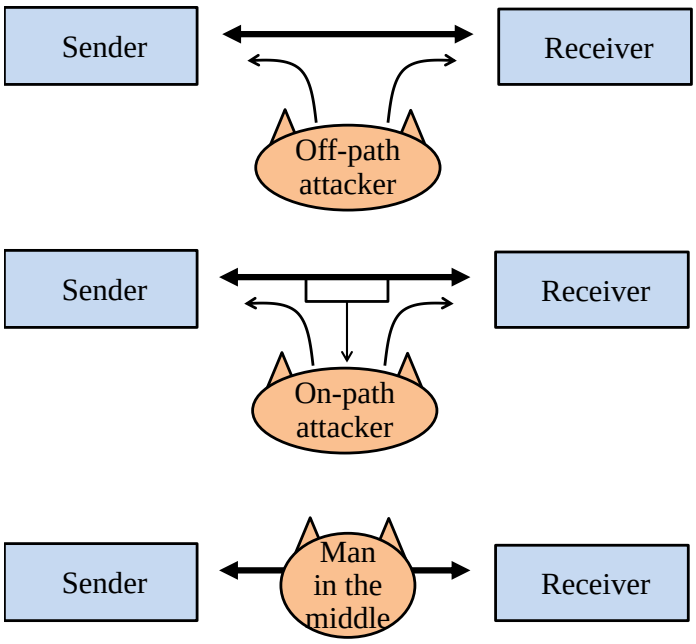

Figure 5·2: Threat models.

prevent time synchronization by dropping packets. Moreover, an MiTM can also bias time synchronization by delaying packets (Mizrahi, 2012a), (Mizrahi, 2012b).[1]

Taking a step back, our work can be seen as a case study of the security risks that arise when network protocols are underspecified. It also highlights the importance of handling diverse protocol requirements in separate and rigourously tested codepaths. Finally, our network protocol analysis introduces new ways of reasoning about network attacks on time synchronization protocols.

5.1.3 Related work

Secure protocols. Our design and analysis of secure client/server protocols comple-

[1]This follows because time-synchronization protocols use information about the delay on the network path in order to accurately synchronize clocks (Section 5.2). A client cannot distinguish the delay on the forward path (from client to server) from the delay on the reverse path (from server to client). As such, the client simply takes the total round trip time δ (forward path + reverse path), and assumes that delays on each path are symmetric. The MiTM can exploit this by making delays asymmetric (*e.g.,* causing the delay on the forward path to be much longer than delay on the reverse path), thus biasing time synchronization.

ment recent efforts to cryptographically secure NTP and its "cousin" PTP (Precision Time Protocol) (Mizrahi, 2012b). Our interest is in securing the core datagram protocol used by NTP, which was last described in David Mills' book (Mills, 2011). To the best of our knowledge, the security of the core NTP datagram protocol has never previously been analyzed. Meanwhile, our analysis assumes that parties correctly distribute cryptographic keys and use a secure MAC. A complementary stream of works propose protocols for distributing keys and performing the MAC, beginning with the Autokey protocol in RFC5906 (Haberman and Mills, 2010), which was broken by Rottger (Röttger, 2012), which was followed by NTS (Sibold et al., 2015), ANTP (Dowling et al., 2016), other works including (Itkin and Wool, 2016), (Moreira et al., 2015), and on-going activity in the IETF (Franke et al., 2018).

Attacks. Our analysis of the NTP specification is motivated, in part, by discovery of over 30 ntpd CVEs between June 2015 to July 2016 (Stenn, 2016). These implementation flaws allow remote code execution, DoS attacks, and timeshifting attacks. Earlier, Selvi (Selvi, 2014), (Selvi, 2015) demonstrated MiTM timeshifting attacks on 'simple NTP (SNTP)' (rather than full-fledged NTP). Even earlier, work (Klein, 2013), (Mills, 2011), (Corbixgwelt, 2011) considered the impact of timeshifting on the correctness of other protocols. The recent academic work (Malhotra et al., 2016) also attacks NTP, but our attacks are stronger. (Malhotra et al., 2016) presented attacks that are on-path (weaker than our off-path attacks), or off-path DoS attacks (weaker than our timeshifting attacks), or off-path time-shifting attacks that needed special client/server configurations (our Zero-0rigin Timestamp attack works in default mode). Also, our measurements find millions of vulnerable clients, while (Malhotra et al., 2016) finds thousands. Finally, NTP's broadcast mode is outside our scope; see (Malhotra and Goldberg, 2016), (Franke et al., 2018), (Mizrahi, 2012b) instead.

Measurement. Our work is also related to studies measuring the NTP ecosystem

Figure 5·3: Timestamps induced by the server response packet (mode 4).

T_1:	*Origin timestamp.*	Client's local time when sending query.
T_2:	*Receive timestamp.*	Server's local time when receiving query.
T_3:	*Transmit timestamp.*	Server's local time when sending response.
T_4:	*Destination timestamp.*	Client's local time when receiving response.

(in past decades) (Minar, 1999), (Murta et al., 2006), the use of NTP for DDoS amplification attacks (Czyz et al., 2014a), the performance of NIST's timeservers (Sherman and Levine, 2016), and network latency (Durairajan et al., 2015). Our attack surface measurements are in the same spirit as those in (Malhotra et al., 2016), (Malhotra and Goldberg, 2016), but we use a new set of NTP control queries. We also provide updated measurements on the presence of cryptographically-authenticated NTP associations.

5.2 NTP Background

NTP's default mode of operation is a hierarchical *client/server* mode. In this mode, timing queries are solicited by clients from a set of servers; this set of servers is typically static and configured manually. *Stratum i* systems act as servers that provide time to stratum $i + 1$ systems, for $i = 1, ...15$. Stratum 1 servers are at the root of the NTP hierarchy. Stratum 0 and stratum 16 indicate failure to synchronize. Client/server packets are not authenticated by default, but a Message Authentication Code (MAC) can optionally be appended to the packet. NTP operates in several additional modes. In *broadcast mode*, a set of clients listen to a server that broadcasts timing information. In *symmetric mode*, peers exchange timing information (Appendix B). There is also an *interleaved mode* for more accurate timestamping (Appendix A.2).

NTP's client/server protocol consists of a periodic two-message *exchange*. The client sends the server a query (*mode 3*), and the server sends back a response (*mode 4*). Each exchange provides a *timing sample*, which uses the four timestamps in Figure 5·3. All four timestamps are 64 bits long, where the first 32 bits are seconds

elapsed since January 1, 1970, and the last 32 bits are fractional seconds. T_1, T_2, and T_3 are fields in the server response packet (mode 4) shown in Figure 5·4. The delay δ is an important NTP parameter (Mills et al., 2010) that measures the round trip time between the client and the server:

$$\delta = (T_4 - T_1) - (T_3 - T_2) \tag{5.1}$$

If there are symmetric delays on the forward and reverse network paths, then the difference between the server and client clock is $T_2 - (T_1 + \frac{\delta}{2})$ for the client query, and $T_3 - (T_4 - \frac{\delta}{2})$ for the server response. Averaging, we get *offset* θ:

$$\theta = \tfrac{1}{2}\left((T_2 - T_1) + (T_3 - T_4)\right) \tag{5.2}$$

A client does *not* immediately update its clock with the offset θ upon receipt of a server response packet. Instead, the client collects several timing samples from each server by completing exchanges at infrequent *polling intervals* (on the order of seconds or minutes). The length of the polling interval is determined by an adaptive randomized *poll process* (Mills et al., 2010, Sec. 13). The poll p is a field on the NTP packet, where (Mills et al., 2010) allows $p \in \{4, 5, .., 17\}$, which corresponds to a polling interval of about 2^p (*i.e.,* 16 seconds to 36 hours).

Once the client has enough timing samples from a server, it computes the jitter ψ. First, it finds the offset value θ^* corresponding to the sample of lowest delay δ^* from the eight most recent samples, and then takes jitter ψ as

$$\psi^2 = \tfrac{1}{k-1}\sum_{i=1}^{k}(\theta_i - \theta^*)^2 \tag{5.3}$$

Typically, $4 \leq k \leq 8$. A client considers updating its clock only if it gets a stream of k timing samples with low delay δ and jitter ψ. This is called `TEST11`.[2]

[2]A single server response packet is sufficient to set time on a SNTP ("simple NTP") client, but

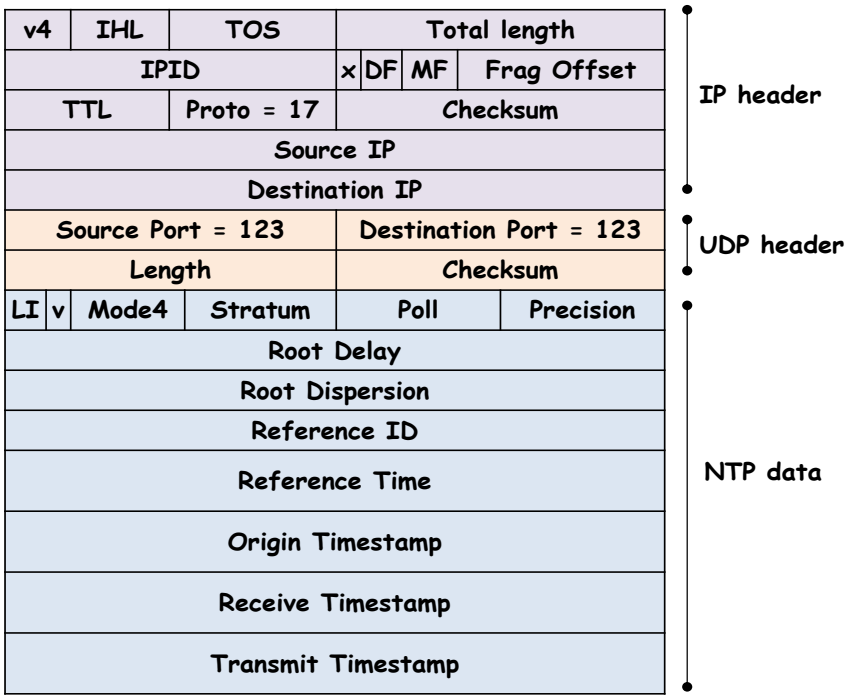

Figure 5·4: NTP server response packet (mode 4). (Client queries have the same format, but with mode field set to 3. Symmetric mode uses mode 1 or 2. Broadcast mode uses mode 5).

TEST11: Check that the *root distance* Λ does not exceed MAXDIST = 1.5 seconds. Λ is proportional [3]to:

$$\Lambda \propto \psi + (\delta^* + \Delta)/2 + E + 2^{\rho} \tag{5.4}$$

The root delay Δ, root dispersion E and precision ρ are from fields in the server's mode 4 response packet (Figure 2·1). Precision ρ is the quality of the system's local clock; $\rho = 12$ implies $2^{-12} = 244\mu$s precision.

After each exchange, the client chooses a *single* server to which it synchronizes its local clock. This decision is made adaptively by a set of *selection*, *cluster*, *combine* and *clock discipline* algorithms (Mills et al., 2010, Sec. 10-12). Importantly, these algorithms can also decide *not* to update the client's clock; in this case, the clock

a stream of self-consistent packets is required for full NTP.

[3]The exact definition of Λ differs slightly between RFC5905 (Mills et al., 2010, Appendix A.5.5.2) and the latest version of ntpd.

runs without input from NTP.

Implementation vs. Specification. RFC 5905 (Mills et al., 2010) specifies NTP version 4, and its "reference implementation" is ntpd (Stenn, 2015d). David Mills, the inventor of NTP, explains (Mills, 2011) the "relationship between the published standard and the reference implementation" as follows: "It is tempting to construct a standard from first principles, submit it for formal verification, then tell somebody to build it. Of the four generations of NTP, it did not work that way. Both the standard and the reference implementation were evolved from an earlier version... Along the way, many minor tweaks were needed in both the specification and implementation..." For this reason, we consider both ntpd and the specification in RFC 5905.

5.3 The Client/Server Protocol in RFC 5905

We now argue that the client/server datagram protocol in RFC 5905 is underspecified and flawed. RFC 5905 mentions the protocol in two places: in its main body (Section 8) and in a pseudo-code listing Appendix A. Because the two mentions are somewhat contradictory, we begin with an overview of the components of NTP's datagram protocol, and then present its specification in Appendix A (Mills et al., 2010, Sec. 8), and in the prose of (Mills et al., 2010, Sec. 8).

5.3.1 Components of NTP's datagram protocol.

NTP uses the *origin timestamp* field of the NTP packet to prevent off- and on-path attacks. (Recall from Figure 5·2 that an off-path attacker can spoof IP packets but cannot eavesdrop on its target's NTP traffic, while an on-path attacker can eavesdrop, inject, spoof, and replay packets, but cannot drop, delay, or tamper with legitimate traffic.) Whenever a client queries its server, the client records the query's sending time T_1 in a local *state variable* (Mills et al., 2010) named "`xmt`". The client then sends T_1 in the *transmit timestamp* of its client query (Figures 5·3 and 5·4). Upon

```
1  receive()
2      if (pkt.T3 == 0 or   # fail test3
3           pkt.T3 == org): # fail test1
4          return
5
6      synch = True
7      if !broadcast:
8          if pkt.T1 == 0:  # fail test3
9              synch = False
10         elif pkt.T1 != xmt:  # fail test2
11             synch = False
12
13     org = pkt.T3
14     rec = pkt.time_received
15     if (synch):
16         process(pkt)
```

Figure 5·5: Pseudocode for the receive function, RFC 5905 Appendix A.5.1.

receipt of the query, the server learns T_1 and copies it into the *origin timestamp* field of its server response (Figure 5·4). When the client receives the server response, it performs `TEST2`:

`TEST2`: The client checks that the origin timestamp T_1 on the server response matches the client's time upon sending the query, as recorded in the client's local state variable `xmt`.

The origin timestamp is therefore a nonce that the client must check (with `TEST2`) before it accepts a response.[4] An off-path attacker cannot see the origin timestamp (because it cannot observe the exchange between client and server), and thus has difficulty spoofing a server response containing a valid origin timestamp. Indeed, the origin timestamp looks somewhat random to the off-path attacker. Specifically, its first 32 bits are seconds, and the last 32 bits are subseconds (or fractional seconds). The first 32 bits appear slightly random because the off-path attacker does not know

[4]Note that ntpd does not randomize the UDP source port to create an additional nonce; instead, all NTP packets have UDP source port 123.

the exact moment that the client sent its query; indeed, Appendix A of RFC 5905 has a comment that says "While not shown here, the reference implementation randomizes the poll interval by a small factor" and the current ntpd implementation randomizes the polling interval by 2^{p-4} seconds when poll $p > 4$. Moreover, the last 32 bits also appear somewhat random because RFC 5905 requires a client with a clock of precision ρ randomize the $(32 - \rho)$- lowest-order bits of the origin timestamp.

The origin timestamp thus is analogous to source port randomization in TCP/UDP, sequence number randomization in TCP, *etc.* When NTP packets are cryptographically authenticated with a MAC, this nonce also provides some replay protection: even an on-path attacker cannot replay a packet from an earlier polling interval because its origin timestamp is now stale.

NTP also has mechanisms to prevent replays within the same polling interval. These are needed because an NTP client continuously listens to network traffic, even when it has no outstanding (*i.e.,* unanswered) queries to its servers. Whenever a client receives a server response packet, it records the transmit timestamp field from the packet in its `org` state variable. The client uses the following test to reject duplicate server response packets:

`TEST1:` The client checks that the transmit timestamp field T_3 of the server response is *different* from the value in the client's `org` state variable.

The client deals with the duplicates of the client's query as follows:

Clear `xmt`: If a server response passes `TEST2`, the client sets its local `xmt` state variable to zero.

Suppose the server receives two identical client queries. The server would send responses to both (because NTP servers are stateless (Mills et al., 2010)). If the client cleared `xmt` upon receipt of the first server response, the second server response packet will be rejected (by `TEST2`) because its origin timestamp is non-zero. At this point,

one might worry that an off-path attacker could inject a packet with origin timestamp set to zero. But, `TEST3` should catch this:

`TEST3`: Reject any response packet with origin, receive, or transmit timestamp T_1, T_2, T_3 set to zero.

5.3.2 Query replay vulnerability in Appendix A of RFC 5905.

Pseudocode from Appendix A of RFC 5905 (see Figure 5·5) handles the processing of received packets of *any* mode, including server mode packets (mode 4), broadcast mode packets (mode 5), and symmetric mode packets (mode 1 or 2). Importantly, this pseudocode requires a host to always listen to and process incoming packets. This is because some NTP modes (*e.g.,* broadcast) process unsolicited packets, and RFC 5905 suggest that all modes use the same codepath. We shall see that this single codepath creates various security problems.

On-path query replay vulnerability. The pseudocode in Figure 5·5 is vulnerable to replays of the client's query. Suppose a client query is replayed to the server. Then, the server will send two responses, each with a valid origin timestamp field (passing `TEST2`) and each with a different transmit timestamp field (passing `TEST1`). The client will accept both responses. Our experiments show that replays of the client query harm time synchronization; see Appendix C.

5.3.3 Zero-0rigin timestamp vulnerability in RFC 5905 prose.

Meanwhile, we find the following in Section 8 of RFC 5905:

> Before the xmt and org state variables are updated, two sanity checks are performed in order to protect against duplicate, bogus, or replayed packets. In the exchange above, a packet is duplicate or replay if the transmit timestamp t3 in the packet matches the org state variable T3. A packet is bogus if the origin timestamp t1 in the packet does not match the xmt state variable T1. In either of these cases, the state variables are updated, then the packet is discarded. To

> protect against replay of the last transmitted packet, the xmt state variable is set to zero immediately after a successful bogus check.

This text describes `TEST1` and `TEST2`, but what does it mean to update the state variables? Comparing this to the pseudocode in Appendix A of RFC 5905 (Figure 5·5 lines 13-14) suggests that this means updating `org` and `rec` upon receipt of any packet (including a bogus one failing `TEST2`), but not the `xmt` state variable.[5] Next, notice that the quoted text does not mention `TEST3`, which rejects packets with a zero-0rigin timestamp. Thus, we could realize the quoted text as pseudocode by deleting lines 8-9 of Figure 5·5. Finally, notice that the quote suggests clearing `xmt` if a received packet passes `TEST2`. Thus, we could add the following after line 11 of Figure 5·5 (with lines 8-9 deleted):

```
else:  xmt = 0
```

However, if `xmt` is cleared but `TEST3` is not applied, we have:

Zero-0rigin Timestamp Attack. The zero-0rigin timestamp vulnerability allows an off-path attacker to hijack an unauthenticated client/server association and shift time on the client.

The attacker sends its target client a spoofed server response packet, spoofed with the source IP address of the target's server.[6] The spoofed server response packet has its origin timestamp T_1 set to zero, and its other timestamps T_2, T_3 set to bogus values

[5] Indeed, suppose we did update the `xmt` variable even after receipt of a bogus packet that fails `TEST2`, with the bogus origin timestamp in the received packet. In this case, we would be vulnerable to a *chosen-origin-timestamp attack*, where an attacker injects a first packet with an origin timestamp of the their choosing. The injected packet fails `TEST2` and is dropped, but its origin timestamp gets written to the target's local `xmt` variable. Then, the attacker injects another packet with this same origin timestamp, which passes `TEST2` and is accepted by the target.

[6] As observed by (Malhotra et al., 2016), hosts respond to unauthenticated mode 3 queries from arbitrary IP addresses by default. The mode 4 response (Figure 5·4) has a *reference ID* field that reveals the IPv4 address of the responding host's time server. Thus, our off-path attacker sends its target a (legitimate) mode 3 query, and receives in response a mode 4 packet, and learns the target's server from its reference ID. Moreover, if the attacker's shenanigans cause the target to synchronize to a different server, the attacker can just learn the IP of the new server by sending the target a new mode 3 query. The attacker can then spoof packets from the new server as well.

designed to convince the client to shift its time. The target will accept the spoofed packet as long is it does not have an outstanding query to its server. Why? If a client has already received a valid server response, the valid response would have cleared the client's `xmt` variable to zero. The spoofed zero-0rigin packet is then subjected to `TEST2`, and its origin timestamp (which is set to zero) will be compared to the `xmt` variable (which is also zero). `TEST3` is never applied, and so the spoofed zero-0rigin packet will be accepted.

Suppose that the attacker wants to convince the client to change its clock by x years. How should the attacker set the timestamps on its spoofed packet? The origin timestamp is set to $T_1 = 0$ and the transmit timestamp T_3 is set to the bogus time now $+ x$. The destination timestamp T_4 (not in the packet) is now $+ d$, where d is the latency between the moment when the attacker sent its spoofed packet and the moment the client received it. Now, the attacker needs to choose the receive timestamp T_2 so that the delay δ is small. (Otherwise, the spoofed packet will be rejected because it fails `TEST11` (Section 5.2).) Per equation (5.1), if the attacker wants delay $\delta = d$, then T_2 should be:

$$T_2 = \delta + T_3 - (T_4 - T_1) = d + \text{now} + x - (\text{now} + d + 0) = x$$

The offset is therefore $\theta = x - \frac{d}{2}$. If the attacker sends the client a stream of spoofed packets with timestamps set as described above, their jitter ϕ is given by the small variance in d (since x is a constant value). Thus, if the attacker sets root delay Δ, root dispersion E and precision ρ on its spoofed packets to be tiny values, the packet will pass `TEST11` and be accepted. This vulnerability is actually present in the current version of ntpd. We discuss how we executed it (CVE-2015-8138) against ntpd in Appendix A.1.

5.4 Leaky Control Queries

Thus far, we have implicitly assumed that the timestamps stored in a target's state variables are difficult for an attacker to obtain from off-path. However, we now show how they can be learned from off-path via NTP control queries. Interestingly, the control queries we use are not mentioned at all in the latest NTP specification in RFC5905 (Mills et al., 2010). However, they are specified in detail in Appendix B of the obsolete RFC1305 (Mills, 1992) from 1992, and are also specified in a new IETF Internet draft (Mills and Haberman, 2016). They have been part of ntpd since at least 1999.[7] NTP's UDP-based control queries are notorious as a vector for DDoS amplification attacks (Czyz et al., 2014a), (Krämer et al., 2015). These DoS attacks exploit the *length* of the UDP packets sent in response to NTP's mode 7 `monlist` control query, and sometimes also NTP's mode 6 `rv` control query. Here, however, we will exploit their *contents*.

The leaky control queries. We found control queries that reveal the values stored in the `xmt` (which stores T_1 per Figure 5·3) and `rec` (which stores T_4) state variables. First, launch the `as` control query to learn the association ID that a target uses for its server(s). (Association ID is a randomly assigned number that the client uses internally to identify each server (Mills, 1992).) Then, the query `rv assocID org` reveals the value stored in `xmt` (*i.e.,* expected origin timestamp T_1 for that server). Moreover, `rv assocID rec` reveals the value in `rec` (*i.e.,* the destination timestamp T_4 for the target's last exchange with its server).

Off-path timeshifting via leaky origin timestamp. If an attacker could continuously query its target for its expected origin timestamp (*i.e.,* the `xmt` state variable), then all bets are off. The off-path attacker could spoof bogus packets that pass `TEST2` and shift time on the target. This is CVE-2015-8139.

[7] `https://github.com/ntpsec/ntpsec/blob/PRE_NT_991015/ntpq/ntpq.c`

Off-path timeshifting attack via interleaved pivot. NTP's interleaved mode is designed to provide more accurate time synchronization. Other NTP modes use the 3-bit *mode* field in the NTP packet (Figure 5·4) to identify themselves (*e.g.,* client queries use mode 3 and server responses use mode 4). The interleaved mode, however, does not. Instead, a host will *automatically* enter interleaved mode if it receives a packet that passes *Interleaved* ***TEST2****. Interleaved* ***TEST2*** checks that the packet's *origin timestamp* field T_1 matches `rec` state variable, which stores T_4 from the previous exchange. Importantly, there is no codepath that allows the host to exit interleaved mode. Appendix A.2 shows that this leads to an extremely low-rate DoS attack that works even in the absence of leaky control queries. This is CVE-2016-1548.

Now consider an off-path attacker that uses NTP control queries to continuously query for `rec`. This attacker can shift time on the client by using its knowledge of `rec` to (1) spoof a single packet passing 'interleaved `TEST2`' that pivots the client into interleaved mode, and then (2) spoof a stream of self-consistent packets that pass 'interleaved `TEST2`' and contain bogus timing information. We have confirmed that this attack works on ntpd v4.2.8p6.

Recommendation: Block control queries! By default, ntpd allows the client to answer control queries sent by any IP in the Internet. However, in response to `monlist`-based NTP DDoS amplification attacks, best practices recommend configuring ntpd with the `noquery` parameter (Stenn, 2015d). While `noquery` should block all control queries, we suspect that `monlist` packets are filtered by middleboxes, rather than by the `noquery` option, and thus many "patched" systems remain vulnerable to our attacks. Indeed, the openNTPproject's IPv4 scan during the week of July 23, 2016 found 705,183 unique IPs responding to `monlist`. Meanwhile, during the same week we found a staggering 3,964,718 IPs responding to the `as` query.[8] The control queries we exploit likely remain out of firewall blacklists because (1) they are

[8]To avoid being blacklisted, we refrained from sending `monlist` queries.

```
rv ‘associd’
rv ‘associd’ org
rv ‘associd’ rec
rv
mode 3 NTPv4 query
```

undocumented in RFC5905 and (2) are thus far unexploited. As such, we suggest that either (1) `noquery` be used, or (2) firewalls block *all* mode 6 and mode 7 NTP packets from unwanted IPs.

5.5 Measuring the Attack Surface

We use network measurements to determine the number of IPs in the wild that are vulnerable to our off-path attacks. We start with zmap (Durumeric et al., 2013) to scan the entire IPv4 address space (from July 27 - July 29, 2016) using NTP's `as` control query and obtain responses from 3,964,718 unique IPs. The scan was broken up into 254 shards, each completing in 2-3 minutes and containing 14,575,000 IPs. At the completion of each shard, we run a script that sends each responding IP the sequence of queries shown below.

These queries check for leaky origin and destination timestamps, per Section 5.4, and also solicit a regular NTP server response packet (mode 4). Our scan did not modify the internal state of any of the queried systems. We solicit server responses packets using RFC 5905-compliant NTP client queries (mode 3), and RFC 1305-compliant mode 6 control packets identical to those produced by the standard NTP control query program ntpq. We obtained a response to at least one of the control queries from 3,822,681 (96.4%) of the IPs responding to our `as` scan of IPv4 address space. We obtained server response packets (mode 4) from 3,274,501 (82.6%) of the responding IPs. Once the entire scan completed on July 29, 2016, we identified all the stratum 1 servers (from the `rv` and mode 4 response packets), and send each the NTP control

Table 5.1: Hosts leaking origin timestamp.

Total	unauthenticated	Stratum 2-15	good timekeepers
3,759,832	3,681,790	2,974,574	2,484,775

query `peers` using ntpq; we obtained responses from 3,586 (76.6%) IPs out of a total of 4,683 IPs queried. (We do this to check if any stratum 1 servers have symmetric peering associations, since those that do could be vulnerable to our attacks.)

5.5.1 State of crypto.

The general wisdom suggests that NTP client/server communications are typically not cryptographically authenticated; this follows because (1) NTP uses pre-shared symmetric keys for its MAC, which makes key distribution cumbersome (NIST, 2010), and (2) NTP's Autokey (Haberman and Mills, 2010) protocol for public-key authentication is widely considered to be broken (Röttger, 2012). We can use our scan to validate the general wisdom, since `as` also reveals a host's 'authentication status' with each of its servers or peers. Of 3,964,718 IPs that responded to the `as` command, we find merely 78,828 (2.0%) IPs that have *all* associations authenticated. Meanwhile, 3,870,933 (97.6%) IPs have *all* their associations unauthenticated. We find 93,785 (2.4%) IPs have at least one association authenticated. For these hosts, off-path attacks are more difficult but not infeasible (especially if *most* of the client's associations are unauthenticated, or if the authenticated associations provide bad time, *etc.*).

5.5.2 Leaky origin timestamps.

Of 3,964,718 IPs responding to the `as` query, a staggering 3,759,832 (94.8%) IPs leaked their origin timestamp. (This is a significantly larger number than the 705,183 IPs that responded to a `monlist` scan of the IPv4 space by the openNTPproject during the same week, suggesting that many systems that have been 'patched' against NTP

DDoS amplification (Czyz et al., 2014a), (Krämer et al., 2015) remain vulnerable to our leaky-origin timestamp attack.)

But how many of these leaky hosts are vulnerable to off-path timeshifting attacks described in Section 5.4? Our results are summarized in Table 5.1. First, we find that only 78,042 (2.1%) of the IPs that leak `org` to us have authenticated *all* associations with their servers, leaving them out of the attackable pool. Next, we note that stratum 1 hosts are not usually vulnerable to this attack, since they sit a the root of the NTP hierarchy (see Section 5.2) and thus don't take time from any server. The only exception to this is the stratum 1 servers that have symmetric peering associations. Combining data from `rv` and mode 3 responses, we find the stratum of the remaining 3,681,790 (97.9%) leaky IPs. We combine this information with the output of the `peers` command, which reveals the 'type of association' each host uses with its servers and peers. Of the 4,608 (0.1%) stratum 1 servers, *none* have symmetric peering associations. Thus, we do not find *any* vulnerable stratum 1 servers.

On the other hand, there are 2,974,574 (80.8%) stratum 2-15 IPs that leak their origin timestamp and synchronize to at least one unauthenticated server. These are all vulnerable to our attack. We do not count 601,043 (16.3%) IPs that have either (1) stratum 0 or 16 (unsynchronized), OR (2) conflicting stratums in `rv` and server responses (mode 4). Finally, we check if these 3M vulnerable IPs are 'functional' or are just misconfigured or broken systems by using data from our mode 3 query scan to determine the quality of their timekeeping. We found that 2,484,775 (83.5%) of these leaky IPs are good timekeepers—their absolute offset values were less than 0.1 sec.[9] Of these, we find 490,032 (19.7%) IPs with stratum 2. These are good targets for attack, so that the impact of the attack trickles down the NTP stratum hierarchy.

[9]We compute the offset θ using equation (6.2), with T_1, T_2, T_3 from the packet timestamps and T_4 from the frame arrival time of the mode 4 response packet .

Table 5.2: Hosts leaking zero-0rigin timestamp.

Total	unauthenticated	Stratum 2-15	good timekeepers
1,269,265	1,249,212	892,672	691,902

Table 5.3: Hosts leaking `rec` and zero-0rigin timestamps. (Underestimates hosts vulnerable to the interleaved pivot timeshifting attack.)

Total	unauthenticated	Stratum 2-15	good timekeepers
1,267,628	1,247,656	893,979	691,393

5.5.3 Zero-0rigin timestamp vulnerability.

The zero-0rigin timestamp vulnerability was introduced seven years ago in ntpd v4.2.6 (Dec 2009), when a line was added to clear `xmt` after a packet passes `TEST2`.[10](This is Line 18 in Figure A·1.) Thus, one way to bound the attack surface for the zero-0rigin timestamp vulnerability is to use control queries as measurement side-channel. We consider all our origin-timestamp leaking hosts, and find the ones that leak a timestamp of zero. Of 3,759,832 (94.8%) origin-timestamp leaking IPs, we find 1,269,265 (33.8%) IPs that leaked a zero-0rigin timestamp. We scrutinize these hosts in Table 5.2 and find $\approx 700K$ interesting targets. Importantly, however, that this is likely an underestimate of the attack surface, since the zero-0rigin vulnerability does *not* require the exploitation of leaky control queries.

5.5.4 Interleaved pivot vulnerability.

The interleaved pivot DoS vulnerability (Appendix A.2)was introduced in the same version as the zero-0rigin timestamp vulnerability. Thus, the IPs described in Section 5.5.3 are also vulnerable to this attack.

[10]See Line 1094 in `ntp_proto.c` in `https://github.com/ntp-project/ntp/commit/fb8fa5f6330a7583ec74fba2dfb7b6bf62bdd246`.

Next, we check which IPs are vulnerable to the interleaved pivot *timeshifting attacks* (Section 5.4). These hosts must (1) leak the `rec` state variable and (2) use a version of ntpd later than 4.2.6. Leaks of `rec` are also surprisingly prevalent: 3,724,465 IPs leaked `rec` (93.9% of the 4M that responded to `as`). These could be vulnerable if they are using ntpd versions post v4.2.6. We cannot identify the versions of all of these hosts, but we do know that hosts that also leak zero as their expected origin timestamp are using versions post v4.2.6. We find 1,267,265 (34%) such IPs and scrutinize them in Table 5.3.

5.6 Securing the Client/Server Protocol.

We now move beyond identifying attacks and *prove security* for modified client/server datagram protocols for NTP.

5.6.1 Protocol descriptions.

Our protocol. Figure 5·6,5·7 present our new client/server protocol that provides 32-bits of randomization for the origin timestamp used in `TEST2`.

Clients use the algorithm in Figure 5·6 to process received packets. While the client continues to listen to server response packets (mode 4) even when it does not have an outstanding query, this receive algorithm has several features that differ from RFC 5905 (Figure 5·5).First, when a packet passes `TEST2`, we clear `xmt` by setting it to a random 64-bit value, rather than to zero. We also require that, upon reboot, the client initializes its `xmt` values for each server to a random 64-bit value. Second, `TEST2` alone provides replay protection and we eliminate `TEST1` and `TEST3`. (`TEST3` is not needed because of how `xmt` is cleared. Eliminating `TEST3` is also consistent with the implementation in ntpd versions after v4.2.6.)

Clients use the algorithm in Figure 5·7 to send packets. Recall that the first 32 bits of the origin timestamp are seconds, and the last 32 bits are subseconds. First, a

```
def client_receive_mode4( pkt ):

    server = find_server(pkt.srcIP)

    if (server.auth == True and
       pkt.MAC is invalid):
        return            # bad MAC

    if pkt.T1 != server.xmt:
        return            # fail test2

    server.xmt = randbits(64)  # clear xmt
    server.org = pkt.T3        # update state variables
    server.rec = pkt.receive_time()
    process(pkt)
return
```

Figure 5·6: Pseudocode for processing a response. We also require that the `xmt` variable be initialized as a randomly-chosen 64-bit value, *i.e.,* `server.xmt = randbits(64)`, when ntpd first boots.

```
def client_transmit_mode3_e32( precision ):

    r = randbits(precision)
    sleep for r*(2**(- precision)) seconds

    # fuzz LSB of xmt
    fuzz = randbits(32 - precision)
    server.xmt = now ^ fuzz

    # form the packet
    pkt.T1 = server.org
    pkt.T2 = server.rec
    pkt.T3 = server.xmt
    ... # fill in other fields

    if server.auth == True:
        MAC(pkt)    #append MAC

    send(pkt)
return
```

Figure 5·7: This function is run when the polling algorithm signals that it is time to query `server`. If `server.auth` is set, then `pkt` is authenticated with a MAC.

client with a clock of precision ρ put a $(32-\rho)$-bit random value in the $(32-\rho)$ lowest order bits. Next, the client obtains the remaining ρ bits of entropy by randomizing the packet's sending time. When the polling algorithm indicates that a query should be sent, the client sleeps for a random subsecond period in $[0, 2^{-\rho}]$ seconds, and then constructs the mode 3 query packet. We therefore obtain 32 bits of entropy in the expected origin timestamp, while still preserving the semantics of NTP packets—the mode 4 packet's origin timestamp field (Figure 5·4) still contains T_1 (where T_1 is as defined in Figure 5·3).

Notice that this protocol only modifies the client, and is fully backwards-compatible with today's stateless NTP servers:

Stateless server algorithm. Today's NTP servers are stateless, and so do not keep `org` or `xmt` state variables for their clients. Instead, upon receipt of client's mode 3 query, a server immediately sends a mode 4 response packet with (1) origin timestamp field equal to the transmit timestamp field on the query, (2) receive timestamp field

```
def client_transmit_mode3_e64( precision ):

    # store the origin timestamp locally
    server.localxmt = now

    # form the packet
    server.xmt = randbits(64) #64-bit nonce
    pkt.T1 = server.org
    pkt.T2 = server.rec
    pkt.T3 = server.xmt
    ... # fill in other fields

    if server.auth == True:
        MAC(pkt)    #append MAC

    send(pkt)
return
```

Figure 5·8: Alternate client/server protocol used by chronyd/openNTPd, that randomizes all 64-bits of the origin timestamp. This function is run when the polling algorithm signals that it is time to query `server`.

set to the time that the server received the query, and (3) transmit timestamp field to the time the server sent its response.

Chronyd/openNTPd protocol. The chronyd and openNTPd implementations also use a client/server protocol that differs from the one in RFC5905. This protocol just sets the expected origin timestamp to be a random 64-bit nonce (see Figure 5·8). While this provides 64-bits of randomness in the origin timestamp, it breaks the semantics of the NTP packet timestamps, because the server response packet no longer contains T_1 as defined in Figure 5·3. (Instead, the client must additionally retain T_1 in local state variable `server.localxmt`.) This means that the chrony/openNTPd protocol *cannot* be used for NTP's symmetric mode (mode 1/2), but our protocol (which preserves timestamp semantics) can be used for symmetric mode. (See footnote 1.)

Security. Both our protocol (Figures 5·6,5·7) and the chronyd/openNTPd protocol (Figures 5·6,5·8) can be used to protect client/server mode from off-path attacks (when NTP packets are unauthenticated) and on-path attacks (when NTP packets are authenticated with a secure message authentication code (MAC)[11].) Security holds

[11]RFC 5905 specifies MD5(key||message) for authenticating NTP packets, but this is not a secure MAC (Bellare et al., 1996). We are currently in the processes of standardizing a new secure MAC for NTP (Malhotra and Goldberg, 2019).

as long as (1) all randomization is done with a cryptographic pseudorandom number generator (RNG), rather than the weak `ntp_random()` function currently used by ntpd (NTPrand, 2014), (2) the expected origin timestamp is not leaked via control queries, and (3) NTP strictly imposes $k = 4$ or $k = 8$ as the minimum number of consistent timing samples required before the client considers updating its clock. The last requirement is needed because 32-bits of randomness, alone, is not sufficient to thwart a determined attacker. However, by requiring k consistent timing samples in a row, the attacker has to correctly guess about $32k$ random bits (rather than just 32 random bits). Fortunately, because of TEST11 (see Section 5.2), ntpd already requires $k \geq 4$ *most* of the time.

To obtain these results, we first develop a cryptographic model for security against off- and on-path NTP attacks (Section 5.6.2). We then use this model prove security for off-path attacks (Section 5.6.3) and on-path attacks (Section 5.6.4), both for our protocol, and for the chronyd/openNTPd protocol.

5.6.2 Security Model.

Our model, which is detailed in Appendix F, is inspired by prior cryptographic work that designs synchronous protocols with guaranteed packet delivery (Katz et al., 2013a), (Achenbach et al., 2015). However, unlike these earlier models, we consciously omit modeling the more powerful MiTM who can drop, modify, or delay packets (see Section 5.1 and Figure 5·2). We assume instead that the network delivers all packets sent between the ℓ honest parties $\mathcal{P}_1$,...,$\mathcal{P}_\ell$. We also assume that the network does not validate the source IP in the packets it transits, so that the attacker can *spoof* packets. Honest parties experience a delay ϱ before their packets are delivered, but the attacker can win every race condition.

The network orchestrates execution of several NTP exchanges (akin to the 'environment' in the universal composability framework (Canetti, 2001)) through the use

of a *transcript* that stipulates (1) which parties engage in two-message client/server exchanges with each other, (2) when they engage in each exchange, and (3) the times t_c and t_s on the local clocks of the client and server respectively during each exchange. We require security over *all* possible transcripts. This means, as a corollary, that the attacker can choose the optimal transcript for her to attack, including having control over the local clocks of *all* honest parties. An on-path attacker can see every packet sent between honest parties, while an off-path attacker can only see the packet's IP header. (See Figure 5·2.) Clients update their local state which includes (1) the set of servers they are willing to query, (2) the state variables (*e.g.,* $\texttt{xmt}_j$, $\texttt{org}_j$, $\texttt{rec}_j$) for each server $\mathcal{P}_j$, and (3) timing samples from their k most recent exchanges with each server. Then:

Definition 5.6.1 (Soundness (Informal))**.** NTP is (k, ϵ)-*sound* on transcript `ts` if for all *resource-bounded* attackers $\mathcal{A}$ and all parties $\mathcal{P}_i$ who do not query $\mathcal{A}$ as an NTP server, $\mathcal{P}_i$ has k consecutive timing samples from one of its trusted servers that have been modified by $\mathcal{A}$ with probability ϵ. The probability is over the randomness of all parties.

We parameterize by k because NTP has mechanisms that prevent synchronization until the host has a stream of consistent timing samples from a server or peer most likely to represent accurate time. `TEST11` enforces this, for example, by requiring jitter $\psi < 1.5$ seconds. (See Section 5.2).

But how should we parameterize k? One idea is $k = 8$, because `TEST11` depends on the jitter ψ which is computed over at most eight consecutive timing samples (equation 5.3). $k = 8$ is also consistent with pseudocode in Appendix A.5.2 of RFC 5905; this pseudocode describes the algorithm used for clock updates and includes the comment "select the best from the latest eight delay/offset samples". This may be too optimistic though, because we have observed that ntpd v4.2.6 requires only $k = 4$ before it updates its clock upon reboot. ntpd v4.2.8p6 requires only one sample

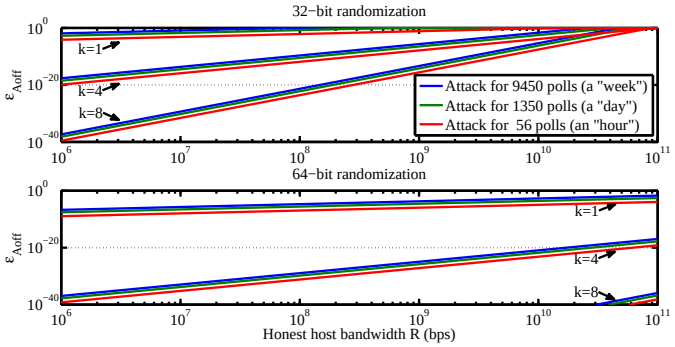

Figure 5·9: Success probability of off-path attacker per Theorem 1: (Top) for Figures 5·6,5·7 and (Bottom) for Figures 5·6,5·8. $\tau \in \{56, 1350, 9540\}$ is the number of polling intervals attacked. We assume one server ($s = 1$) and latencies of at most $\varrho = 1$ second.

upon reboot but this is a bug (CVE-2016-7433); see Appendix D. Thus, we consider $k \in \{1, 4, 8\}$.

5.6.3 Security analysis: Unauthenticated NTP & off-path attacks.

We now discuss the security guarantees for the protocols described in Appendix 5.6. We start by considering off-path attacks. At a high level, our protocol and the chronyd/openNTPd protocols thwart off-path attackers due to the unpredictability of the origin timestamp. Preventing off-path attacks is the best we can hope for when NTP is unauthenticated, since on-path attackers (that can observe the expected origin timestamp per Figure 5·2) can trivially spoof unauthenticated server responses.

We assume that honest parties can send and receive packets at rate at most R bits per second (bps). The network imposes latencies of $\leq \varrho$ for packets sent by any honest party. The polling interval is 2^p, where RFC 5905 constrains $p \leq 17$. Let $\mathit{off}\mathcal{A}$ denote the off-path attackers and let ts be any transcript that involves ℓ honest parties, τ the maximum number of exchanges involving any single client-server pair, k is the minimum number of consistent timing samples needed for a clock update and s the maximum number of trusted servers per client. Also let Adv(RNG) denote the maximum advantage that any attacker with $\mathit{off}\mathcal{A}$'s resource constraints has of

distinguishing a pseudorandom number generator from a random oracle. Then both protocols satisfy the following:

Theorem 1. *Suppose NTP is unauthenticated. Let off$\mathcal{A}$, k, ϱ, p, R, ts, ℓ, s, τ and $Adv(RNG)$ be as described above. Then the protocol Figures 5·6 5·7 is $(k, \epsilon_{off\mathcal{A}})$-sound on transcript ts with*

$$\epsilon_{off\mathcal{A}} = Adv(RNG) + (k+1)s\tau \cdot \left[\frac{2^{-32}kR\varrho}{360}\right]^k \tag{5.5}$$

[12] *And the protocol in Figure 5·6 5·8 is $(k, \epsilon_{off\mathcal{A}})$-sound on transcript ts with*

$$\epsilon_{off\mathcal{A}} = Adv(RNG) + (k+1)s\tau \cdot \left[\frac{2^{p-64}kR}{720}\right]^k \tag{5.6}$$

Figure 5·9 plots $\epsilon_{off\mathcal{A}}$ versus the bandwidth R at honest parties for $k \in \{1, 4, 8\}$ and different values of τ, where τ is the number of polling intervals for which the attacker launches his attack. We do this both for our protocol in Figure 5·7 (Figure 5·9(top)) and the chronyd/openNTPd protocol in Figure 5·8 (Figure 5·9(bottom)).Since hosts typically use a minimum poll value $p_{\min} = 6$, the values $\tau = (9450, 1350, 56)$ in Figure 5·10 correspond to attacking one (week, day, hour) of $2^{p_{\min}} = 64$ second polling intervals. We also assume one server $s = 1$, overestimate network latencies as $\varrho = 1$ second, overestimate poll p in equation (5.6) as $p = 17$. We assume a good RNG so Adv(RNG) is negligible.

$k = 4$ is sufficient with 32-bits of randomness. Recall that k is the minimum number of consistent timing samples needed for a clock update. Figure 5·9(top) indicates that $k = 4$ suffices for our protocol (that randomizes the 32-bit sub-second granularity of

[12] Our soundness definition (Appendix 5.6.2) both allows the off-path attacker to choose the transcript (and thus also the clients local time t_c when it sends the packet) and to see the IP header (only) of the sent packet. Thus, for our protocol (that provides 32-bits of randomness), the off-path attacker essentially knows T_1 up to the second (but not sub-second) granularity. This allows us to claim security even against an off-path attacker that predicts the behavior of a target's polling algorithm (but not her cryptographic random number generators (RNGs)). Some off-path attackers may realistically be able to do this. Consider an off-path attacker that sends a target a 'packet-of-death' that triggers a reboot of ntpd (*e.g.,* CVE-2016-9311 or CVE-2016-7434). Because the attacker knows when ntpd rebooted, it may be able to predict the behavior of its polling algorithm.

the expected origin timestamp). Even if an off-path attacker attacks for a week, his success probability remains less than 0.1% as long as $k = 4$ and the target accepts packets at bandwidth $R = 5$ Gbps or less. When the attacker attacks for an hour, the target's bandwidth must be $R \approx 19$ Gbps for a 0.1% success probability. To put this in context, endhosts typically send < 10 NTP packets per *minute*, and even the large stratum 1 timeservers operated by NIST process queries at an average rate of 21 Gbps (Sherman and Levine, 2016). Therefore, it seems unlikely that an attacker could attack for hours or days without being detected. If more security is needed, we could take $k = 8$, which requires a bandwidth of $R \approx 40$ Gbps for a one-hour attack with success probability of 0.1%. Meanwhile, Figure 5·9(top) suggests that 32-bits of randomness do not suffice to limit off-path attacks when $k = 1$. This should provide further motivation for fixing the ntpd v4.2.8p6 bug that allows $k = 1$ upon reboot (see Appendix D).

$k = 1$ is sufficient with 64-bits of randomness. Meanwhile, Figure 5·10 (bottom) indicates that $k = 1$ suffices for the chronyd/openNTPd protocol that randomizes all 64 bits of the expected origin timestamp. Even if an off-path attacker attacks for a week, his success rate remains less than 0.1% as long as the target's bandwidth is limited to $R = 5$ Gbps. Moreover, when $k = 4$, attacking for a week at 100 Gbps only yields a success probability of 10^{-17}.

5.6.4 Security analysis: Authenticated NTP & on-path attacks.

Both our protocol (Figure 5·6,5·7) and the chronyd/openNTPd protocol (Figure 5·6,5·8) thwart on-path attackers when NTP packets are authenticated with a MAC.

We let sending rate R, network latency ϱ, poll p and Adv(RNG) be as before. Let $on\mathcal{A}$ be an on-path attacker, and let ts be any transcript that involves ℓ honest parties, a maximum of s trusted servers per client and a maximum of τ exchanges involving any single client-server pair that replicate any t_c value (up to the second) at

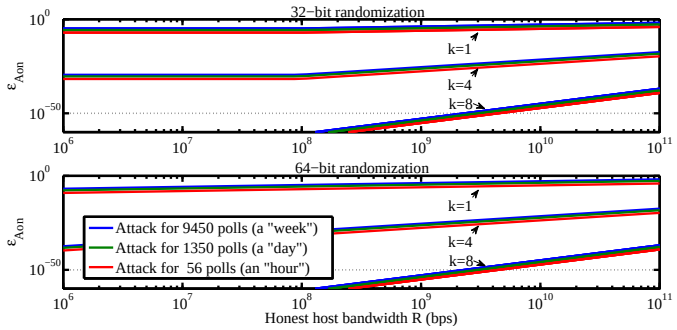

Figure 5·10: Success probability of on-path attacker per Theorem 2: (Top) for Figures 5·6,5·7; in this case we overestimate the number of legitimate client queries that have identical 32 high-order bits of origin timestamp as $\gamma = 100$. (Bottom) for Figures 5·6,5·8. $\tau \in \{56, 1350, 9540\}$ is the number of polling intervals attacked. We assume one server ($s = 1$), latencies of at most $\varrho = 1$ second, MAC of length $2n = 128$ bits and maximum poll value $p = 17$.

most γ times. Let Adv(EU-CMA) be the maximum probability that an attacker with $on\mathcal{A}$'s resource constraints can forge a MAC of length $2n$ under a chosen-message attack. Then both protocols satisfy the following:

Theorem 2. *Suppose NTP is authenticated with a MAC of length* $2n$*. Let* $on\mathcal{A}$*,* k*,* ϱ*,* p*,* R*,* ts*,* ℓ*,* s*,* τ*,* γ*, Adv(EU-CMA) and Adv(RNG) be as described above. Then, both protocols are* $(k, \epsilon_{on\mathcal{A}})$*-sound on transcript* ts *with*

$$\epsilon_{on\mathcal{A}} \leq Adv(RNG) + (k+1)s\tau(kQ)^k, \tag{5.7}$$

where

$$Q = \max\left\{q_E + \frac{R\varrho \cdot Adv(EU\text{-}CMA)}{360 + n}, \frac{2^{p-64}R\varrho}{720 + 2n}\right\}. \tag{5.8}$$

where $q_E = 2^{-32}\gamma$ *for the protocol in Figures 5·6 5·7 and* $q_E = 2^{-64}\tau$ *for the protocol in Figures 5·6 5·8.*

To argue about security, we assume a good MAC (like CMAC (Malhotra and Goldberg, 2019)) so that Adv(EU-CMA) $\approx 2^{-128}$. We overestimate $p = 17$ in equation (5.8) and $\varrho = 1$ second and plot $\epsilon_{on\mathcal{A}}$ versus R for one server ($s = 1$) and different choices of τ in Figure 5·10.

With 32-bits of randomness, $k = 4$ *is sufficient.* Suppose the 32-bits sub-second granularity of the expected origin timestamp is randomized. When R is small, Fig-

ure 5·10(top) indicates that the on-path attacker's success rate is dominated by the first term inside the maximum in equation (5.8). This corresponds to a successful replay attack, because the client has sent multiple queries with the same expected origin timestamp. Meanwhile, when R is large, second term in the maximum in equation (5.8) dominates. This corresponds to a successful replay attack, because the client 'cleared' `xmt` to a random 64-bit value that matches an origin timestamp in an earlier query. Again, the attacker's success probability is disconcertingly high when $k = 1$. [13] On the other hand, excellent security guarantees are obtained for $k = 4$, so it is safer to have $k \geq 4$.

With 64-bits of randomness, $k = 4$ is sufficient. Suppose now that the entire 64-bits of the expected origin timestamp is randomized. Now the second term in the maximum in equation (5.8) always dominates. This again corresponds to a successful replay attack, because the client 'cleared' `xmt` to a random 64-bit value that matches an origin timestamp in an earlier query.

5.7 Summary and Recommendations

We have identified several vulnerabilities in the NTP specifications both in RFC5905 (Mills et al., 2010) and in its control query specification in (obsoleted) RFC1305 (Mills, 1992), leading to several working off-path attacks on NTP's most widely used client/server mode (Section 5.3-5.4). Millions of IPs are vulnerable our these attacks (Section 4.4).

We present denial-of-service attacks on symmetric mode in Appendix B.

Many of our attacks are possible because RFC5905 recommends that same code-path is used to handle packets from all of NTP's different modes. Our strongest attack, the zero-0rigin timestamp attack (CVE-2015-8139), follows because NTP's

[13] The poor results for $k = 1$ and 32-bits of randomization follow because our model allows the 32 high-order bits of the expected origin timestamp to repeat in at most γ different queries. It might be tempting to dismiss this by assuming $\gamma = 0$, but basing security on this is not a good idea. For example, a system might always boot up thinking that it is January 1, 1970.

client/server mode shares the same codepath as symmetric mode. (In Section B.3, we explain why the initialization of symmetric mode requires that hosts accept NTP packets with origin timestamp set to zero; this leads to the zero-0rigin timestamp attack on client/server mode, where the attacker convinces a target client to accept a bogus packet because its origin timestamp is set to zero.) Similarly, the fact that interleave mode and client/server mode shares the same codepath gives rise to the interleave pivot attack (CVE-2016-1548). Thus, we recommend that different codepaths be used for different modes. This is feasible, since a packet's mode is trivially determined by its *mode* field (Figure 5·4). The one exception is interleaved mode, so we suggest that interleaved mode be assigned a distinguishing value in the NTP packet.

Our attacks also follow because the NTP specification does not properly respect `TEST2`. We therefore propose a new backwards-compatible client/server protocol that gives `TEST2` the respect it deserves (Section 5.6.1). We developed a framework for evaluating the security of NTP's client/server protocol and used it to prove that our protocol prevents (1) off-path spoofing attacks on unauthenticated NTP and (2) on-path replay attacks when NTP is cryptographically authenticated with a MAC. We have proved the similar results for a different client/server protocol used by chronyd and openNTPD. (See Section 5.6.3,5.6.4.) We recommend that implementations adopt either protocol.

Our final recommendation is aimed at systems administrators. We suggest that firewalls and ntpd clients block *all* incoming NTP control (mode 6,7) and timing queries (mode 1,2 or 3) from unwanted IPs (Section 5.4), rather than just the notorious `monlist` control query exploited in DDoS amplification attacks.

Chapter 6

Universally Composable Treatment of Network Time

6.1 Introduction

Most existing large-scale networks, and in particular the global Internet, are predominantly asynchronous and do not require the participants to be "synchronized" with other entities in any way or have a global sense of time. In fact, this non-reliance on a common notion of time can be seen as one of the reasons for the success of the TCP/IP design.

However, as it turns out, several important mechanisms that are central to the usability of networks as a platform for communication and distributed computation do indeed require parties to have some global, common sense of real-time. Interestingly, the need for a global sense of time does not arise from the desire to provide synchronous communication, quality of service, or other "sophisticated" networking primitives. Rather, awareness to real time is often coupled with the safe use of cryptography to thwart attacks against the network.

One prevalent use of real time is in revoking, and limiting the duration of certificates for public keys. Indeed, verifying the validity of the public key of one's peer for communication is a crucial step in setting up authenticated communication, which in turn is the basis for practically any security-aware interaction on the Internet today. Setting time limit to the validity of certificates, and furthermore revoking certificates

when necessary, is a crucial component in making Public-Key Infrastructure (PKI) a valid, usable basis for secure communication. Such ability, in turn, hinges on having good sense of current real time. Furthermore, not only mainframe servers need to have such ability – even low end clients need it, in fact arguably even more so than servers. Indeed, without a good sense of current time, a client cannot verify whether a certificate is valid, or whether a given certificate revocation list is the up-to-date one. Other uses of real time to improve security include various forms of timestamping for contracts and timing transactions in public ledgers.

It may appear that measuring real time is a relatively easy task; indeed, most computing platforms today, even low-end ones, are equipped with a built-in clock. Still, synchronizing and adjusting these clocks, and in particular reaching agreement on time in a large, asynchronous network like the Internet turns out to be non-trivial. In particular, NTP, the current IETF standard protocol for computers on the Internet to determine time (Mills et al., 2010), is rather complex. It assumes a hierarchical system of "time servers," where lower-stratum servers are assumed to have a more accurate notion of time, and higher-stratum servers determine time by querying several lower-stratum ones and performing some complex aggregation of the responses. The protocol has mechanisms for protecting from errors introduced by network delays, but is built on complete trust in the queried time servers, as well as in the authenticity of the communication. Indeed, NTP has been demonstrated to be easily subvertible, resulting in massive loss of security (Malhotra et al., 2016), (Malhotra and Goldberg, 2016), (Malhotra et al., 2017), (Czyz et al., 2014b).

Several variants of NTP such as sntpd (Mills, 2006), ptpd (ptpd, 2015), chronyd (chronyd, 2015), OpenNTPD (openNTPD, 2012), ntimed (ntimed, 2015), and Roughtime (roughtime, 2015) have been proposed. These protocols offer varying degree of clock accuracy, correctness, precision and security guarantees. They have different packet

semantics and a different mechanism on how the querying client chooses to update its local time, if at all, after interacting with one or potentially many time servers.

When coming to assess these proposals, it becomes evident that we don't currently have a good measure to test these proposals against. Indeed, while great many analytical works propose ways to model time (either real, global, or relative) within network protocols, and even within security protocols, we do not have a way to rigorously capture the security guarantees from a network time protocol that provably suffice for security-sensitive applications that require an agreed-upon time measurement— for instance for guaranteeing the validity of certificates in a way that, in turn, will guarantee authenticated and secure communication. (See Section 6.1.4 for a brief account of related work on the modeling of time.)

6.1.1 Our Contributions

We provide a modular, composable formalism of the security requirements from network-time protocols — or, more generally from protocols that provide a reading of real time with the assistance of other nodes over an asynchronous network. Specifically, we propose formal abstractions of secure network time, and show that:

- Our abstractions of network-time suffice for securely incorporating expiration times in certificates, as well as freshness guarantees for public certificate lists, in a way that guarantees PKI-based secure communication *even in face of an adversary who tries to subvert the measurement of time and at the same time corrupts revoked and expired certificates.*

- Our abstractions are realizable by simple protocols that mimic the behavior of authenticated NTP.

We use the Universally Composable (UC) security framework as a basis for our formalism. Indeed, the UC framework provides a general mechanism for specifying security

properties of cryptographic protocols in a way that facilitates composing protocols together, and in particular guarantees that composition of secure components results in overall security of the composed protocol. Furthermore, the UC framework is geared towards analyzing the security of cryptographic protocols, which facilitates incorporating the results in this work with existing analytical results for cryptographic protocols.

Specifically, we build upon an existing analytical work by Canetti et al. that asserts, within the UC framework, the security of authentication and key exchange protocols that are based on global public-key infrastructure (PKI) (Canetti et al., 2016). We incorporate our analysis of timing consensus achieved via network time into the UC analysis of a global PKI. The combined analysis extends the security guarantees provided by (Canetti et al., 2016) to the case of revocable and expirable certificates.

Our methodology of incorporating network time into existing UC protocols and functionalities is quite generic. Hence, our work paves the way toward instantiating time consensus and reaping its security benefits within other UC formalisms in a seamless fashion.

Technical and Conceptual Challenges. A priori it appears that the UC framework might be unsuitable for representing real time. Indeed, the framework is centered around modeling completely asynchronous, event-driven systems. Furthermore, in the UC formalism the basic computational elements (Turing machine instances) are activated one by one, and the order of execution and activation of components is under total adversarial control. This is done with good reason, namely in order to provide security even against adversaries that have full control over the network; however, this structure appears to be incompatible with the modeling of real time that advances "at the same rate" within all components of a physically spread-out

system. (It should be noted that this asynchronous, event-driven formalism that gives the adversary total control over the scheduling of events is not unique to the UC framework. Indeed, it is the common methodology for modeling and analyzing cryptographic protocols in general — for the same reason outlined above.)

Our first contribution is thus to propose a construct that represents global time even within such a system. The construct is simple: It is a trusted entity (formalized as a global ideal functionality) that keeps a counter. This counter is incremented adversarially by the environment, but is guaranteed to never decrease. All entities in the system have access to this counter (or, rather, some perturbed version of it, as described below) — which they treat as Time. Indeed, this adversarially incremented counter does not in any way approximate the passing of real physical time. Still, we argue that from the point of view of capturing the validity of mechanisms that use time in order to provide some security guarantees, this simple gadget is good enough. Said otherwise, any security property that is expressible and asserted within our formal framework would be preserved even when implemented in a real system that has access to real physical time.

Another set of challenges has to do with the modeling of the "imperfections" that one encounters when using the currently available mechanisms for measuring time. We consider two main methods for measuring time, each with its own imperfections: One method is measuring one's own local physical clock. This method provides fast response and relatively accurate measurement of time elapsed between events that occur at the same location; however, the response may be arbitrarily "shifted" relative to actual real time. The second method is asking one (or more) other entities in the network ("time servers") for their current time reading. This method can potentially provide reading of real time, but is susceptible to measurement errors due to network delays, spoofing attacks, and faulty servers. It may also be slow in

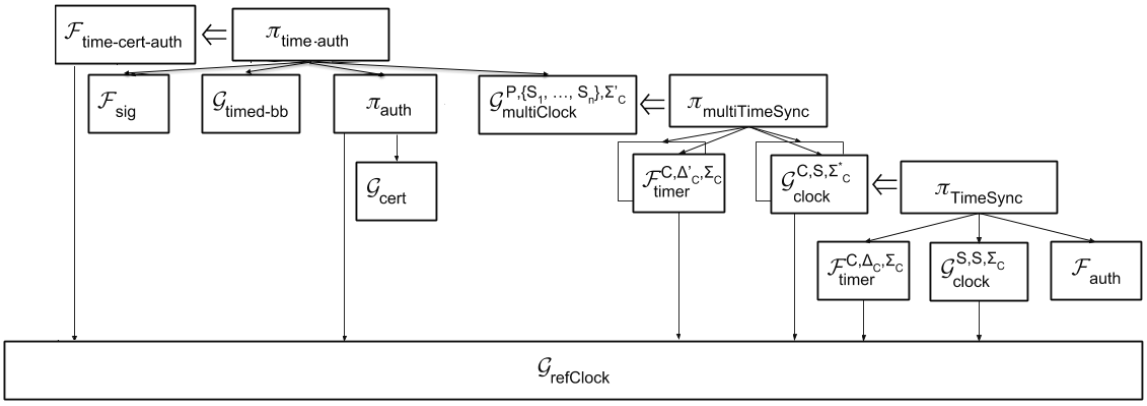

Figure 6·1: Overview of our formalism, from the exact, approximate, and relative time functionalities to ideal certification with limited-time certificates. Double arrows mean "UC realizes," and single arrows mean "uses as a subroutine."

providing a response. Indeed, a good network time protocol is one that combined these two methods in a "secure way" in order to provide a reliable reading of real time. Our goal is to capture that property.

6.1.2 Our Formalism in a Nutshell

We provide a brief overview of our formalism. See Figure 6·1 for the relationships between these primitives.

The GUC framework. Writing a specification within the Global Universal Composability (GUC) framework amounts to writing a program for an *ideal functionality* $\mathcal{F}$ that captures the expected behavior of the analyzed system π. Here $\mathcal{F}$ captures both the expected functionality and the expected security properties. Formally, system π is said to GUC-realize $\mathcal{F}$ if for any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that no external environment $\mathcal{E}$ can tell whether it is interacting with $\mathcal{A}$ and π or with $\mathcal{S}$ and $\mathcal{F}$. Here $\mathcal{E}$ plays the role of a calling protocol that provides inputs to π (or $\mathcal{F}$) and obtains the outputs of π (or $\mathcal{F}$), whereas $\mathcal{A}$ controls the communication between the parties running π. The communication between $\mathcal{S}$ and $\mathcal{F}$ captures the

"security imperfections" that $\mathcal{F}$ allows.

The main added feature of the GUC framework beyond the original UC framework is that it allows incorporating in the model of execution "global functionalities" that represent trusted services that exist in the system regardless of the analyzed protocol. That is, the global functionalities exist both in the ideal model for functionality $\mathcal{F}$ and in the model for executing π. This modeling allows to better capture long-term services such as public-key infrastructure (as done in (Canetti et al., 2016)), or network time — as done here.

Exact and Approximate Clocks. Our first basic construct is a global ideal functionality $\mathcal{G}_{\text{refClock}}$ that provides an exact clock. Formally, it provides a non-decreasing counter that the environment can increment at will. In a sense, the clock's idealistic time serves as a reference or benchmark to which everyone aspires, even though none of the parties directly interacts with $\mathcal{G}_{\text{refClock}}$ itself.

Instead, parties only interact with $\mathcal{G}_{\text{refClock}}$ indirectly through a timer functionality and a network clock functionality that provide *approximate* relative and absolute notions of time, respectively. The timer functionality $\mathcal{F}_{\text{timer}}$ captures a cheap but low-latency device that can only provide measurements locally without delay, and the measurements "drift" significantly.

The network clock $\mathcal{G}_{\text{clock}}$ provides information more globally, to all parties in the system; however it may not respond to queries right away (or ever!). This functionality captures the behavior expected from a single client-server execution of the network time protocol, since the adversary controls the delays of packets transmitted over the asynchronous network. On the plus side, $\mathcal{G}_{\text{clock}}$ guarantees that timing measurements are approximately accurate (up to some bound) at the moment that they are eventually given.

Realizing $\mathcal{G}_{\text{clock}}$. We provide two network protocols that realize the network clock functionality. The first protocol π_{timeSync} involves a single query-response exchange between a client C with has access to a local timer (i.e., an instance of $\mathcal{F}_{\text{timer}}$) and a server S that has access to her own clock. This protocol allows the client to "bootstrap" the server's clock into one of her own, as long as the server is honest (i.e., uncorrupted).

We also capture a generalization of $\mathcal{G}_{\text{clock}}$ whose accuracy depends on multiple servers in such a way that it is robust to the corruption of a few servers. This generalization, denoted $\mathcal{G}_{\text{multiClock}}$, allows a client to request time from multiple servers (each with their own $\mathcal{G}_{\text{clock}}$) and then to select time as a function of all responses obtained before its $\mathcal{F}_{\text{timer}}$ times out (e.g., by picking the median response). By selecting the time in this way, the client obtains resilience against network corruptions. Even if many of the sessions are compromised, the client's timing measurement approximates the reference time as long as the majority of the servers whose $\mathcal{G}_{\text{clock}}$ boxes responded quickly to a query were uncorrupted, akin to the "sleepy model" of consensus (Pass and Shi, 2016), (Micali, 2016).

PKI with Expiration and Revocation. The final piece in our formalism is a time-aware variant of public-key infrastructure and signature verification. The starting point of our formalism is the certified signature verification functionality $\mathcal{G}_{\text{cert}}$ of (Canetti et al., 2016) that utilizes infinite-duration keys. However, since the guarantee provided by $\mathcal{G}_{\text{cert}}$ has no time limit, it follows that $\mathcal{G}_{\text{cert}}$ cannot be realized in a system where signature keys get compromised after some time has elapsed.

In this work, we extend $\mathcal{G}_{\text{cert}}$ by allowing each signer to provide an expiration time t^*; furthermore, the signer can update this time in order to emulate revocation. Our extended functionality guarantees that if the global time at the time of verification is larger than t^* plus some "fudge factor" that accounts for the inaccuracies in time

measurement, then the verification necessarily fails. This "fudge factor" is of crucial importance: it determines the length of time for which certificate authorities must respond to CRL or OCSP queries about certificates after they expire.

6.1.3 Additional Discussion

Incorporating Time in Existing UC Modeling, a General Paradigm. Our method for incorporating the time constraints in $\mathcal{G}_{\text{cert}}$ and in the protocol that realizes it minimal and general: To obtain time-aware certification, we only add a simple, self-contained time check to the existing code of to $\mathcal{G}_{\text{cert}}$. Additionally, we observe that the security of timing-agnostic protocols is unaffected by the presence of time-sensitive protocols.

Putting together these two observations, we obtain a general methodology for adding time-sensitive protocols and functionalities to the existing UC framework and its corpus of secure functionalities in a seamless way.

Time is Global. We model time as a global construct. That is, the time-related functionalities $\mathcal{G}_{\text{refClock}}$, $\mathcal{G}_{\text{clock}}$, and $\mathcal{G}_{\text{multiClock}}$ are global: they are accessible by anyone in the system. In particular, they always exist both in the "ideal" and in the "real" system. This modeling simplifies the composition of protocols that use these joint functionalities and provide a closer modeling of reality. We choose to model $\mathcal{F}_{\text{timer}}$ as a local functionality since it represents a service that is available only locally to a party. However this functionality too can in principle be modeled as a global functionality. (Such modeling might indeed be useful for analyzing systems where several protocols that use time have access to the same local physical clock.)

Implementing Authenticated Communication. Our network time protocols rely upon authenticity of communication between C and S. When specifying the

protocols, we assume the existence of an ideal authenticated communication functionality $\mathcal{F}_{\text{auth}}$ (Canetti, 2004) as specified in Fig. 6·2, and we use the modularity of the framework to remain agnostic about $\mathcal{F}_{\text{auth}}$'s underlying implementation.

Functionality $\mathcal{F}_{\text{auth}}$

1. Upon receiving $(\mathsf{Send}, \mathsf{sid}, B, m)$ from party A, send $(\mathsf{Sent}, \mathsf{sid}, A, B, m)$ to $\mathcal{A}$.
2. Upon receiving $(\mathsf{Send}, \mathsf{sid}, B', m')$ from the $\mathcal{A}$, do: If A is corrupted then output $(\mathsf{Sent}, \mathsf{sid}, A, m')$ to party B'. Else, output $(\mathsf{Sent}, \mathsf{sid}, A, m)$ to party B. Halt.

Figure 6·2: The authenticated communication functionality, $\mathcal{F}_{\text{auth}}$. Reproduced from (Canetti, 2004).

We can instantiate $\mathcal{F}_{\text{auth}}$ using a PKI-based authenticated communication, as in, e.g., (Canetti et al., 2016). But we intend to use time to bolster the PKI! Ergo, we must avoid circularity in our arguments.

One way to do so is to assume that time servers have certificates that do not expire, or else where revocation is done out-of-band. Alternatively can instantiate $\mathcal{F}_{\text{auth}}$ as NTP does: have the client and server use an out-of-band key exchange mechanism to perform symmetric key authentication. Specifically, the maintainer of most stratum 1 NTP servers, NIST, shares keys with its clients over U.S. mail or fax machines (NIST, 2018). This method completely circumvents the reliance on PKI for realizing $\mathcal{F}_{\text{auth}}$. Potentially, there are other out-of-band mechanisms for key exchange such as biometric human identification.

6.1.4 Related Work

The ability for parties to obtain a notion of time is an integral part of distributed computations (Lamport, 1978). These computations often require that timing measurements satisfy specific properties depending on the nature of the computation.

The most basic of these is that time be monotonically increasing to allow for a consistent and correct ordering of events in, e.g., a time stamping protocol (Haber and Stornetta, 1991), (Matsuo and Matsuo, 2005), (Buldas et al., 2005). However, many protocols such as those of (Goldreich, 2006) and (Kalai et al., 2005) require stronger guarantees, namely that time is both synchronized between parties and advances in a relatively uniform and expected pace.

A number of formalisms have been proposed over the years for incorporating time (both absolute and relative) in the security analysis of protocols. Some of these formalisms, like ours, are based on the UC framework (Kalai et al., 2005), (Backes et al., 2014), (Vajda, 2016), (Katz et al., 2013b), (Canetti, 2013). We briefly review them.

UC Analyses of Time. The modeling of time by (Kalai et al., 2005) is perhaps the closest to the one in this work. There too, time is modeled as an additional "counter" that is available to all machines, and is incremented by the adversary on each machine, individually, subject to some global constraints. However, there it is assumed that all parties have ideal access to the global time (and furthermore that communication delays are within known time bounds.) In fact our work can be viewed as a more detailed and "faithful" modeling of the propagation of time in real-life networks, in a way justifying the more abstract modeling of (Kalai et al., 2005). In other words, if one assumes that a majority of the instances of $\mathcal{G}_{\text{clock}}^{P,S,\Sigma}$ used by the parties are uncorrupted, (Kalai et al., 2005) can be used as an additional application for our modeling. Furthermore, the impossibility result in (Kalai et al., 2005) implies that one cannot realize $\mathcal{G}_{\text{clock}}^{P,S,\Sigma}$ or $\mathcal{G}_{\text{refClock}}$ with appropriate parameters in the plain model.

(Katz et al., 2013b) and (Canetti, 2013) provide, within the UC framework, ideal functionalities that give abstractions that mimic synchronous communication among

participants. However, these works do not provide ways for realizing these abstractions from existing mechanisms like network time. Furthermore, these abstractions do not suffice to capture the prevalent use of limited-time certificates.

(Backes et al., 2014) provide an alternative formalism of time based on the UC framework that differs from the present formalism in a number of crucial ways. First, (Backes et al., 2014) significantly modifies the existing UC framework, thus making its formalism incompatible with the body of work in the existing framework. Second, the (Backes et al., 2014) modeling assumes that machines have specific and fixed relative speeds and where time passage is directly proportional to the number of computational steps. In contrast, in our modeling time is not necessarily tied to other computational aspects of the system. Third, (Backes et al., 2014) analyzes only standard, time-unaware protocols; the modeling of time is used only to bound the success of attacks on the protocol. In contrast, we model protocols where time is crucially used by the protocol itself.

(Vajda, 2016) provides a number of high-level proposals for general modeling of real-time within the UC framework. However this work does not address network time protocols or the cryptographic applications treated here.

Security-Aware Network Time Protocols. Several frameworks (Malhotra et al., 2017), (Dowling et al., 2016), (Mizrahi, 2012b), (Itkin and Wool, 2016) aim to define and analyze the security requirements of time synchronization protocols. RFC 7384 (Mizrahi, 2012b) provides guidelines for important security features of PTP and NTP as they relate to possible attacks. (Itkin and Wool, 2016) build on this with new attack vectors and suggested mitigations for PTP, but they do not provide proofs for their mitigations nor any accuracy guarantees for the time protocols themselves.

(Dowling et al., 2016) extend NTP to include lightweight authentication for servers and provide game based proofs for its accuracy relative to the time at the

server. They do not provide any accuracy guarantees relative to a global notion of time, and thus they fail to provide the global synchronization that is necessary for time sensitive crypto such as PKI.

(Malhotra et al., 2017) focus on several security concerns when deploying NTP in practice, at the expense of full coverage. They study concrete security bounds for NTP against off/on-path attacks in the standalone model. By contrast, the security guarantee in our work addresses composable security and additionally covers adversaries who have full control over the network and may use it to drop packets sent by honest parties. Both of these properties are crucial towards our work in Section 6.6 of realizing time-sensitive crypto primitives like the PKI.

6.1.5 Organization

The paper is organized as follows. Section 6.2 gives an overview of the Network Time Protocol. In Section 6.3, we introduce three new ideal functionalities: $\mathcal{G}_{\text{refClock}}$, $\mathcal{G}_{\text{clock}}^{P,S,\Sigma}$, and $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$. Section 6.4 formalizes and proves the security of a protocol that permits a single server to share its view of time with a single client. Section 6.5 generalizes this basic protocol in two ways for improved resilience: a client takes timing measurements from multiple servers, and the network topology is dispersed to reduce resource and network resource congestion. Finally, Section 6.6 integrates our time consensus protocol with time-sensitive applications such as PKI.

6.2 Preliminaries

This section summarizes (separately!) the universally composable security framework and the network time protocol.

6.2.1 Universally Composable Security

We provide a brief overview of the UC framework. See (Canetti, 2013) and (Canetti et al., 2007) for more details. (This overview is taken almost verbatim from (Canetti et al., 2016).)

We focus on the notion of protocol *emulation*, wherein the objective of a protocol π is to imitate another protocol ϕ. In this work, the entities and protocols we consider are polynomial-time bounded Interactive Turing Machines (ITMs), in the sense detailed in (Canetti, 2013).

Systems of ITMs. To capture the mechanics of computation and communication among entities, the UC framework employs an extension of the ITM model. A computer program (such as for a protocol, or perhaps program of the adversary) is modeled in the form of an ITM. An execution experiment consists of a system of ITMs which are instantiated and executed, with multiple instances possibly sharing the same ITM code. A particular executing ITM instance running in the network is referred to as an ITI. Individual ITIs are parameterized by the program code of the ITM they instantiate, a party ID (pid) and a session ID (sid). We require that each ITI can be uniquely identified by the identity pair id = (pid,sid), irrespective of the code it may be running. All ITIs running with the same code and session ID are said to be a part of the same protocol session, and the party IDs are used to distinguish among the various ITIs participating in a particular protocol session.

The Basic UC Framework. At a very high level, the intuition behind security in the basic UC framework is that any adversary $\mathcal{A}$ attacking a protocol π should learn no more information than could have been obtained via the use of a simulator $\mathcal{S}$ attacking protocol ϕ. Furthermore, we would like this guarantee to hold even if ϕ were to be used as a subroutine in arbitrary other protocols that may be running

concurrently in the networked environment and after we substitute π for ϕ in all the instances where it is invoked. This requirement is captured by a challenge to distinguish between actual attacks on protocol ϕ and simulated attacks on protocol π. In the model, attacks are executed by an environment $\mathcal{E}$ that also controls the inputs and outputs to the parties running the challenge protocol. The environment $\mathcal{E}$ is *constrained* to execute only a single instance of the challenge protocol. In addition, the environment $\mathcal{E}$ is allowed to interact freely with the attacker (without knowing whether it is $\mathcal{A}$ or $\mathcal{S}$). At the end of the experiment, the environment $\mathcal{S}$ is tasked with distinguishing between adversarial attacks perpetrated by $\mathcal{A}$ on the challenge protocol π, and attack simulations conducted by $\mathcal{S}$ with protocol ϕ acting as the challenge protocol instead. If no environment can successfully distinguish these two possible scenarios, then protocol π is said to *UC-emulate* the protocol ϕ.

Balanced environments. In order to keep the notion of protocol emulation from being unnecessarily restrictive, we consider only environments where the amount of resources given to the adversary (namely, the length of the adversary's input) is at least some fixed polynomial fraction of the amount of resources given to all protocols in the system. From now on, we only consider environments that are balanced.

Definition 1 (UC-emulation)**.** Let π and ϕ be multi-party protocols. We say that π UC-emulates ϕ if for any adversary $\mathcal{A}$ there exists an adversary $\mathcal{S}$ such that for any (constrained) environment $\mathcal{E}$, we have:

$$\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}} \approx \text{EXEC}_{\phi,\mathcal{S},\mathcal{E}}$$

Defining protocol execution this way is sufficient to capture the entire range of network activity that is observable by the challenge protocol but may be under adversarial control.

Furthermore, the UC framework admits a very strong composition theorem, which

guarantees that arbitrary instances of ϕ that may be running in the network can be safely substituted with any protocol π that UC-emulates it. That is, given protocols ρ, π and ϕ, such that ρ uses subroutine calls to ϕ, and protocol π UC-emulates ϕ, let $\rho^{\phi \to \pi}$ be the protocol which is identical to ρ except that each subroutine call to ϕ is replaced by a subroutine call to π. We then have:

Theorem 3 (UC-Composition)**.** *Let ρ,π and ϕ be protocols such that ρ makes subroutine calls to ϕ. If π UC-emulates ϕ and both π and ϕ are subroutine-respecting, then protocol $\rho^{\phi \to \pi}$ UC-emulates protocol ρ.*

The Generalized UC Framework. As mentioned above, the environment $\mathcal{E}$ in the basic UC experiment is unable to invoke protocols that share state in any way with the challenge protocol. In contrast, in many scenarios we would like to be able to analyze challenge protocols that share information with other network protocol sessions. For example, protocols may share information via a global setup such as a public Common Reference String (CRS) or a standard Public Key Infrastructure (PKI). To overcome this limitation and allow analyzing such protocols in a modular way, (Canetti et al., 2007) propose the Generalized UC (GUC) framework. The GUC challenge experiment is similar to the basic UC experiment, only with an *unconstrained* environment. In particular, now $\mathcal{E}$ is allowed to invoke and interact with arbitrary protocols, and even multiple sessions of the challenge protocol. Some of the protocol sessions invoked by $\mathcal{E}$ may even share state information with challenge protocol sessions, and indeed, those protocol sessions might provide $\mathcal{E}$ with information related to the challenge protocol instances that it would have been unable to obtain otherwise. To distinguish this from the basic UC experiment, we denote the output of an unconstrained environment $\mathcal{E}$, running with an adversary $\mathcal{A}$ and a challenge protocol π in the GUC protocol execution experiment, by $\text{GEXEC}_{\pi,\mathcal{A},\mathcal{E}}$. GUC emulation is defined analogously to the definition of basic UC emulation outlined above:

Definition 2 (GUC-emulation)**.** Let π and ϕ be multi-party protocols. We say that π GUC-emulates ϕ if for any adversary $\mathcal{A}$ there exists an adversary $\mathcal{S}$ such that for any (unconstrained) environment $\mathcal{E}$, we have:

$$\mathrm{GEXEC}_{\pi,\mathcal{A},\mathcal{E}} \approx \mathrm{GEXEC}_{\phi,\mathcal{S},\mathcal{E}}.$$

The UC theorem directly extends to the GUC model.

6.2.2 The Network Time Protocol (NTP)

As the name suggests, NTP permits several computers on a network to share information about the time.

In this work, we focus on NTP's most popular method of operation: a hierarchical client-server fashion in which a client queries a server who has (ostensibly) higher fidelity timing information than the client. A client can use multiple invocations of NTP's fundamental query-response protocol (either with the same server or with multiple servers) to gather several timing measurements, which it then uses to set or update its own notion of time.

Query-Response Protocol. We first describe NTP's two-round timing exchange protocol over IPv4. Four timing measurements are relevant during the execution of this protocol:

T_1 *Origin timestamp.* Client's system time at the moment that the client sends the query.

T_2 *Receive timestamp.* Server's system time at the moment that the server receives the query.

T_3 *Transmit timestamp.* Server's system time at the moment that the server sends the response.

T_4 *Destination timestamp.* Client's system time at the moment that the client receives the response.

The client's query packet includes measurement T_1. The server's response packet repeats T_1 and appends T_2 and T_3. The client locally computes T_4 upon receipt of the response packet.

Setting, or Updating, the Client's Time. The client makes two assumptions when analyzing the timestamps.

1. Its clock and the server's clock move in relative synchrony while the NTP session is live (even if they have different absolute notions of time).

2. The network delay is symmetric. That is, the query packet's client $\rightarrow$ server latency equals the response packet's server $\rightarrow$ client latency.

Deviations from these assumptions do lead to small but bounded error in the client's eventual measurement of time; we will return to this issue later.

If assumption 1 is accurate, then the round-trip network *delay* δ during the exchange equals:

$$\delta = (T_4 - T_1) - (T_3 - T_2) \tag{6.1}$$

If assumption 2 is accurate, then the absolute gap between the server and client clock is $T_2 - (T_1 + \frac{\delta}{2})$ for the client query, and $T_3 - (T_4 - \frac{\delta}{2})$ for the server response. Averaging these two quantities gives us the absolute *offset* between the client and server clocks:

$$\theta = \tfrac{1}{2}\left((T_2 - T_1) + (T_3 - T_4)\right) \tag{6.2}$$

While talking to multiple servers, the client chooses a single server to which it synchronizes its local clock. This decision is made adaptively by a set of selection, cluster, combine and clock discipline algorithms. For the purpose of this paper, we

assume that the client will make an update to its clock if median θ is less than a certain threshold. Client/server packets are not authenticated by default, but a Message Authentication Code (MAC) can optionally be appended to the packet (Mills et al., 2010, Sec. 13). In this work, we restrict our attention to authenticated NTP.

6.3 Modeling Absolute and Relative Time

In this section, we introduce three ideal functionalities that aid a client C or server S to learn the time. The first two provide exact and approximate *absolute* notions of time, whereas the third functionality approximates the *relative* passage of time. These functionalities are depicted formally in Figs. 6·4 through 6·5. We stress that the functionalities only respond to the methods explicitly stated in the figures; when given a message that cannot be parsed into one of the provided forms, they simply hand the execution back to the caller without providing any output.

The functionalities themselves are referred to as $\mathcal{G}_{\text{functionality}}^{\text{paramaters}}$ for global functionalities and as $\mathcal{F}_{\text{functionality}}^{\text{paramaters}}$ for local (i.e., non-global) functionalities. Protocols are specified in the format $\pi_{\text{functionality}}^{\text{paramaters}}$. The protocols and functionalities may use subroutines, which we indicate in the text and sometimes denote using square brackets.

6.3.1 The Reference Clock Functionality $\mathcal{G}_{\text{refClock}}$

We begin by introducing a simple global functionality $\mathcal{G}_{\text{refClock}}$ that provides a universal *reference clock*. When queried, it provides an abstract notion of time represented as an integer G. It is monotonic, and only the environment may increment it; we stress that the simulator $\mathcal{S}$ *cannot* forge the reference time.

In this work we use subscripts to denote the relative order in which requests are made to the reference clock. Hence, if $x > y$ then $\mathsf{G}_x \geq \mathsf{G}_y$.

Figure 6·4 formally codifies $\mathcal{G}_{\text{refClock}}$. It functions similarly to Vadja's ideal notion of time (Vajda, 2016), with one crucial exception: we do not intend for any honest

Functionality $\mathcal{G}_{\text{clock}}^{P,S,\Sigma}$ [$\mathcal{G}_{\text{refClock}}$]

A clock functionality identified by a session id $\mathsf{sid}_{\mathsf{clock}} = (\mathsf{sid}'_{\mathsf{clock}}, P, S)$ that denotes its owner P as well as a (potentially but not necessarily different) party S whose honesty influences the accuracy of the clock. It is also parameterized by the maximum allowable shift Σ from the reference time. It operates as follows.

Corrupt: Upon receiving a $\mathsf{Corrupt}$ message, record that S is now corrupted.

GetTime: Upon receiving input $(\mathsf{GetTime}, \mathsf{sid}_{\mathsf{clock}})$ from party P', ignore this request if $P' \neq P$, otherwise:

1. Send $(\mathsf{Sleep}, \mathsf{sid}_{\mathsf{clock}})$ to the adversary $\mathcal{A}$. Wait for a response of the form $(\mathsf{Wake}, \mathsf{sid}_{\mathsf{clock}}, \sigma)$ from $\mathcal{A}$.
2. If $\sigma == \perp$, output $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{clock}}, \perp)$ to P.
3. Else send $\mathsf{GetTime}$ to $\mathcal{G}_{\text{refClock}}$ to receive G. Next, compute $T_P = \mathsf{G} + \sigma$. Then do the following:
   - If $|\sigma| > \Sigma$ and no $\mathsf{Corrupt}$ record exists, then reset $T_P = \perp$.
   - Output $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{clock}}, T_P)$ to P.

Figure 6·3: Ideal functionality $\mathcal{G}_{\text{clock}}^{P,S,\Sigma}$ that provides delayed, approximately accurate time measurements to its owner P. The functionality doesn't provide any guarantee on when a timing measurement will be delivered. It only guarantees that at the instant the measurement is given, its value is approximately correct.

party to access $\mathcal{G}_{\text{refClock}}$ directly. Instead, in this work $\mathcal{G}_{\text{refClock}}$ exclusively functions as a subroutine for the remaining two functionalities.

6.3.2 Delayed Approximate Clock Functionality $\mathcal{G}_{\mathbf{clock}}^{P,S,\Sigma}$

In Figure 6·3, we construct the global functionality $\mathcal{G}_{\text{clock}}^{P,S,\Sigma}$ that provides *delayed, approximate time*. This functionality has three major distinctions from $\mathcal{G}_{\text{refClock}}$.

First, the clock communicates with a single party P, who we refer to as its owner, and the accuracy of the clock can be influenced by (potentially but not necessarily different) party S. This degree of freedom allows us to use the clock functionality in

Functionality $\mathcal{G}_{\text{refClock}}$

$\mathcal{G}_{\text{refClock}}$ maintains an integer G corresponding to the reference time. When created, the box initializes $\mathsf{G} = 0$.

IncrementTime: Upon receiving an IncrementTime request from $\mathcal{E}$, update $\mathsf{G} \leftarrow \mathsf{G} + 1$ and send an ok message to $\mathcal{E}$. Ignore IncrementTime requests sent by any other entity.

GetTime: Upon receiving a GetTime command, return G to the calling entity.

Figure 6·4: Global Ideal functionality representing reference time $\mathcal{G}_{\text{refClock}}$. It is expected that honest parties do not talk to this functionality directly.

this work as an abstraction of two very different situations: (1) a physical clock that is actually under the control of its owner, such as an atomic clock owned by a stratum 1 NTP server, and (2) an ideal service akin to that expected from the Network Time Protocol itself, which permits a client to operate "as if" she owned a clock herself, modulo the unavoidable imperfections (cf. Theorems 4-5).

Second, the clock is inaccurate, in the sense that its belief about the time $T = \mathsf{G}+\sigma$ is somewhat shifted away from the reference time. Still, the clock is guaranteed to *approximate* the reference time up to some maximum shift value $|\sigma| \leq \Sigma$.

Third, the clock is *not* instantaneous. Instead, it only returns a time measurement after some adversarially-controlled *delay* δ (which may be infinite). The approximate correctness guarantee from above holds at the moment that the time is *eventually* returned.

$\mathcal{G}_{\text{clock}}^{P,S,\Sigma}$ is used both as a goal, or *benchmark*, in the sense that everyone strives to attain it (with the best parameters possible), and at the same time it is used as a service for other protocols in order to achieve other tasks (or even another instance of $\mathcal{G}_{\text{clock}}^{P,S,\Sigma}$ but with better parameters or another server S).

6.3.3 Approximate Timer Functionality $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$

In Figure 6·5, we construct the $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$ functionality. Like local clocks, a timer functionality has a single owner C and its measurements only guarantee approximate correctness. However, $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$ differs from the prior two functionalities in three ways. First, it doesn't provide an absolute notion of time; instead, it provides the *relative* difference in time between a starting and ending point.

Second, the timer has a short lifetime: after a maximum delay Δ_C, it will "time out" and only output $\perp$. This limitation ensures that $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$ cannot be used as a substitute for long-term time measurements of the type provided by $\mathcal{G}_{\text{refClock}}$ and $\mathcal{G}_{\text{clock}}^{P,S,\Sigma}$.

Third, it is only used locally within a specific instance of a time synchronization protocol. By contrast, $\mathcal{G}_{\text{refClock}}$ and $\mathcal{G}_{\text{clock}}^{P,S,\Sigma}$ are Global UC functionalities.

6.4 Single Server Time Sync

In this section, we formally specify a simplified version of the way that a client C uses the Network Time Protocol (NTP) to query a single server S for its belief about the time. We show that this protocol $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ allows C to operate as if she had a delayed approximate clock of her own. Formally, we prove that $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ GUC-realizes a clock $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$ owned by the client.

As depicted in Fig. 6·7, $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ internally uses the functionalities specified in Section 6.3. The server has access to its own instance of $\mathcal{G}_{\text{clock}}^{S,S,\Sigma_S}$ that approximates the reference time, whereas the client can only measure the relative passage of time via $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$. Additionally, S and C communicate using $\mathcal{F}_{\text{auth}}$.

We fully specify the protocol $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ in Figure 6·6. The protocol is natural: C sends a time request to S and uses $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$ to measure the time elapsed until the response arrives. S responds with (roughly) the times at which she receives the

Functionality $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$ [$\mathcal{G}_{\text{refClock}}$]

A timer functionality identified by a session id $\mathsf{sid}_{\mathsf{timer}} = (\mathsf{sid}'_{\mathsf{timer}}, C)$ that denotes its owner C. It is also parameterized by the maximum allowable delay Δ_C, and the maximum allowable shift Σ_C. It operates as follows.

SetShift: Upon receiving a command $(\mathsf{SetShift}, \mathsf{sid}_{\mathsf{timer}}, M)$ from $\mathcal{E}$, record M as the code of a Turing machine (replacing any previously-stored code) and send an $(\mathsf{ok}, \mathsf{sid}_{\mathsf{timer}})$ message to $\mathcal{E}$.

Start: Upon receiving input $(\mathsf{Start}, \mathsf{sid}_{\mathsf{timer}})$ from a party P: if $P \neq C$ or if a Start command was previously received then ignore this request. Otherwise:

1. Send GetTime to $\mathcal{G}_{\text{refClock}}$. Denote its response as G'. Record the tuple $(\mathsf{G}', \mathsf{sid}_{\mathsf{timer}})$.
2. Send an $(\mathsf{ok}, \mathsf{sid}_{\mathsf{timer}})$ message to C.

TimeElapsed: Upon receiving input TimeElapsed from a party P: if no previous Start command was issued or if $P \neq C$ then ignore this request. Otherwise:

1. Send GetTime to $\mathcal{G}_{\text{refClock}}$. Denote its response as G. Also, retrieve the previously-recorded G'.
2. Run $M(\mathsf{sid}_{\mathsf{timer}}, \mathsf{G}', \mathsf{G})$ and denote its output as σ_C. (Also, maintain M's state for future calls.)
3. Compute $\delta = \mathsf{G} - \mathsf{G}' + \sigma_C$. If $\delta \leq \Delta_C$, output $(\delta, \mathsf{sid}_{\mathsf{timer}})$ to C. Else output $(\perp, \mathsf{sid}_{\mathsf{timer}})$ to C.

Figure 6·5: Ideal functionality $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$ that returns to its owner C the approximate relative time elapsed between the Start and TimeElapsed commands. While the environment $\mathcal{E}$ may influence the timer's accuracy, it must do so 'out of band': once C requests TimeElapsed, it learns the answer instantaneously. Additionally, the adversary doesn't directly interact with $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$ at all.

Protocol $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ [$\mathcal{F}_{\text{auth}}$, $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$, $\mathcal{G}_{\text{clock}}^{S,S,\Sigma_S}$]

GetTime: Begins when the caller (e.g., $\mathcal{E}$) sends the input $(\mathsf{GetTime}, \mathsf{sid_{ts}})$ to C, where $\mathsf{sid_{ts}} = (\mathsf{sid'_{ts}}, C, S)$. Ends when C responds back to the caller with $(\mathsf{TimeReceived}, \mathsf{sid_{ts}}, T_c)$.

$\mathcal{E}$	Client C	$\mathcal{F}_{\text{auth}}$	Server S	$\mathcal{A}$
$\longrightarrow$	1: Record the tuple $(\mathsf{sid_{ts}}, \mathsf{sid_{timer}})$			
	2: Send Start to $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$, receive $(\mathsf{ok}, \mathsf{sid_{timer}})$			
	3: Send $(\mathsf{query}, \mathsf{sid_{ts}})$ to S	$\longrightarrow$	4: Record the tuple $(\mathsf{sid_{ts}}, \mathsf{sid_{clock}})$	
			5: Send $(\mathsf{GetTime}, \mathsf{sid_{clock}})$ to $\mathcal{G}_{\text{clock}}^{S,S,\Sigma_S}$	$\longleftrightarrow$
			6: Get $(\mathsf{TimeReceived}, \mathsf{sid_{clock}}, T_2)$	
			7: Append T_2 to the record $(\mathsf{sid_{ts}}, \mathsf{sid_{clock}})$	
			8: Send $(\mathsf{Sleep}, \mathsf{sid_{ts}})$ to $\mathcal{A}$	$\longrightarrow$
			9: Get $(\mathsf{Wake}, \mathsf{sid_{ts}})$ from $\mathcal{A}$	$\longleftarrow$
			10: Send $(\mathsf{GetTime}, \mathsf{sid_{clock}})$ to $\mathcal{G}_{\text{clock}}^{S,S,\Sigma_S}$ again	$\longleftrightarrow$
			11: Get $(\mathsf{TimeReceived}, \mathsf{sid_{clock}}, T_3)$	
			12: Retrieve $(\mathsf{sid_{ts}}, \mathsf{sid_{clock}}, T_2)$	
		$\longleftarrow$	13: Send $(\mathsf{response}, \mathsf{sid_{ts}}, T_2, T_3)$ to C	
	14: Retrieve $(\mathsf{sid_{ts}}, \mathsf{sid_{timer}})$, abort if no tuple exists			
	15: Send $\mathsf{TimeElapsed}$ to $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$, receive $(\delta, \mathsf{sid_{timer}})$			
	16: If $\delta == \perp$, then set $T_C = \perp$			
	17: Else, set $T_C = T_3 + \frac{1}{2} \cdot (\delta - T_3 + T_2)$			
	18: Delete record $(\mathsf{sid_{ts}}, \mathsf{sid_{timer}})$			
$\longleftarrow$	19: Output $(\mathsf{TimeReceived}, \mathsf{sid_{ts}}, T_C)$ to $\mathcal{E}$			

Continue: Begins when $\mathcal{A}$ sends a $\mathsf{Continue}$ message to C, so that C can determine whether an ongoing session of $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ has timed out. If so, C ends the session.

$\mathcal{A}$	Client C
$\longrightarrow$	1: Retrieve $(\mathsf{sid_{ts}}, \mathsf{sid_{timer}})$, abort if no tuple exists
	2: Send $\mathsf{TimeElapsed}$ to $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$, receive $(\delta, \mathsf{sid_{timer}})$ in response
$\longleftarrow$	3: If $\delta == \perp$, then delete record $(\mathsf{sid_{ts}}, \mathsf{sid_{timer}})$ and output $(\mathsf{TimeReceived}, \mathsf{sid_{ts}}, \perp)$ to $\mathcal{E}$
$\longleftarrow$	4: If $\delta \neq \perp$, then end activation

Figure 6·6: Time Synchronization Protocol $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$. The two participants C and S communicate through $\mathcal{F}_{\text{auth}}$, and they have access to $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$ and $\mathcal{G}_{\text{clock}}^{S,S,\Sigma_S}$, respectively. Note that S does not have any inputs or outputs.

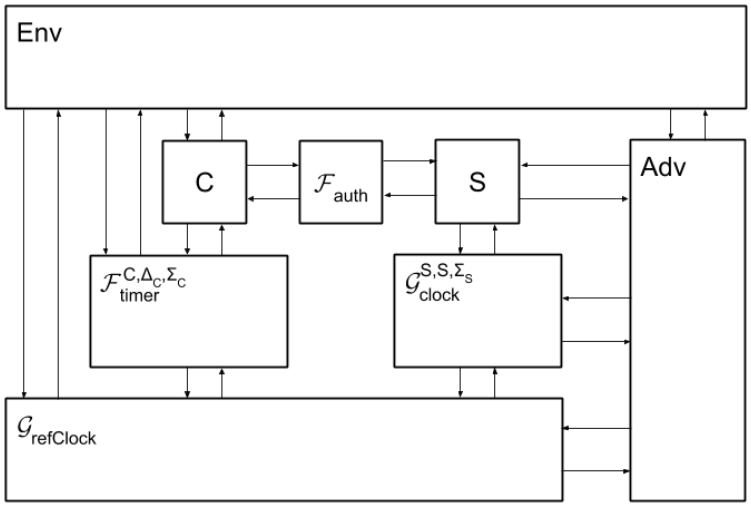

Figure 6·7: Interactions between participants and functionalities during an execution of the single server time synchronization protocol $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$.

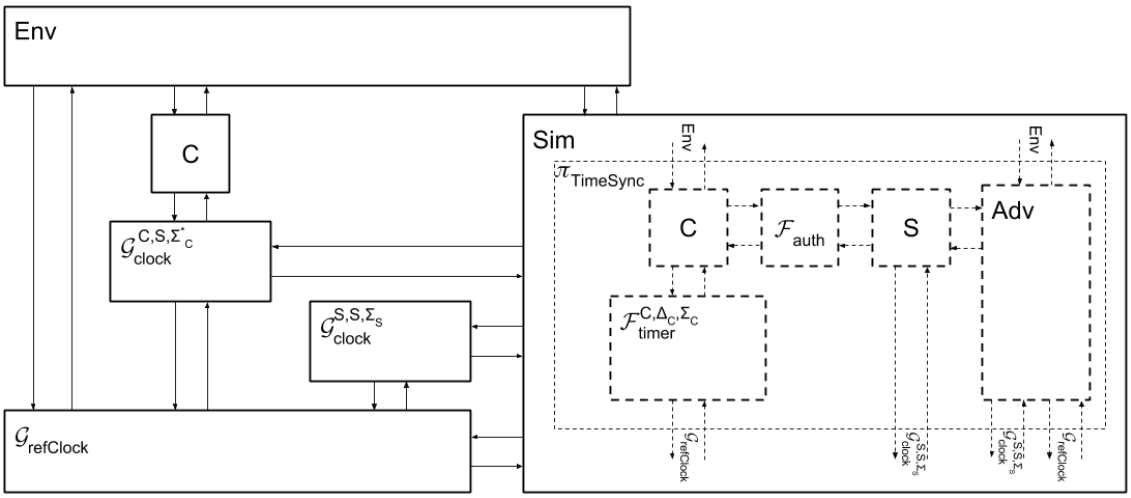

Figure 6·8: Interactions between participants in the ideal world execution of single server time sync, including the emulation of the real world inside of the simulator.

client query (T_2) and sends the response packet (T_3) to allow the client to distinguish network transmission time from server processing time. The client uses its local timer to determine how long the server took to respond as well as to calculate the average network delay in an NTP-like manner; however, the client times out if the server responds after too long a delay (measured on the client's local timer).

For simplicity of exposition, we chose to "hardwire" the identity of the server in the code of both the ideal functionality and the protocol. However this is not essential: our results continue to hold in an alternative model where the identity of the server (or servers, in Section 6.5) is given to the client as a part of the input.

The remainder of this section contains a formal theorem and proof about the accuracy of $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$.

Theorem 4 (Single server UC security)**.** *Given any parameters that satisfy* $\Sigma_C^* \geq \frac{1}{2} \cdot \Delta_C + \Sigma_C + \Sigma_S$*, it holds that the single server approximate time synchronization protocol* $\pi_{timeSync}^{\Delta_C,\Sigma_C,\Sigma_S}$ *GUC-realizes* $\mathcal{G}_{clock}^{C,S,\Sigma_C^*}$.

We emphasize that the *shift* of the client's purported time depends on the *delay* that the client waits for the timing information. Hence, C's insistence upon a maximum delay Δ_C isn't merely a matter of convenience: it affects the accuracy of her notion of time as well.

Our proof has two components. First, in Section 6.4.1 we design a simulator $\mathcal{S}$ that successfully emulates the execution of any real-world adversary $\mathcal{A}$ from the environment $\mathcal{E}$'s point of view. Then, in Section 6.4.2 we analyze the time bound that it achieves.

6.4.1 Designing the Simulator

Fig. 6·8 depicts the high-level interaction between components in the ideal world. As usual, the simulator $\mathcal{S}$ runs an emulated copy of the real world protocol $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ inside its head while also playing the role of $\mathcal{E}$ inside this simulation.

In more detail, $\mathcal{S}$ internally emulates the execution of $\mathcal{A}$, $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$, $\mathcal{F}_{\text{auth}}$ and each of the involved parties. In addition, $\mathcal{S}$ externally instantiates the global $\mathcal{G}_{\text{clock}}^{S,S,\Sigma_S}$ that's called by the protocol (unless it already exists). $\mathcal{S}$ also relays the messages sent from the emulated parties to $\mathcal{G}_{\text{clock}}^{S,S,\Sigma_S}$ and $\mathcal{G}_{\text{refClock}}$, and from $\mathcal{G}_{\text{clock}}^{S,S,\Sigma_S}$ and $\mathcal{G}_{\text{refClock}}$ to the emulated parties.

While conducting this simulation, $\mathcal{S}$ monitors the traffic of its emulated $\mathcal{A}$ and $\mathcal{E}$, along with any messages that $\mathcal{S}$ directly receives from $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$. The messages that $\mathcal{S}$ views/receives causes it to make changes in the ideal world or the emulated world.

Simulating GetTime**:** When $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$ sends a message of the form $(\mathsf{Sleep}, \mathsf{sid}_{\mathsf{clock}})$, then $\mathcal{S}$ instantiates C with the environmentally-provided message $(\mathsf{GetTime}, \mathsf{sid}_{\mathsf{ts}})$.

Simulating Continue**:** When $\mathcal{S}$ observes $\mathcal{A}$ sending a Continue message to the emulated C, it waits to see which of the two possible outcomes occur at C. If C simply ends its activation, then a timeout event has not yet occurred and $\mathcal{S}$ does nothing. If instead C outputs $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{clock}}, \perp)$ then a timeout event has occurred; in this case, $\mathcal{S}$ sends a $(\mathsf{Wake}, \mathsf{sid}_{\mathsf{clock}}, \sigma)$ message to $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$ to cause the dummy C to produce the same output in the ideal world.

Relaying messages to $\mathcal{E}$: When the emulated $\mathcal{A}$ sends a message to the emulated $\mathcal{E}$, $\mathcal{S}$ relays it to the real $\mathcal{E}$. Conversely, $\mathcal{S}$ forwards messages sent by the real $\mathcal{E}$ to the emulated $\mathcal{A}$.

Corrupting the server: When $\mathcal{A}$ sends a Corrupt message in the emulated world, then $\mathcal{S}$ sends a Corrupt message to $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$.

Completing the simulation: When the emulated C sends to $\mathcal{E}$ its output $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{ts}}, T_C)$, then $\mathcal{S}$ knows both *when* and *what* to send back to $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$.

If $T_C == \perp$, then $\mathcal{S}$ simply sends $(\mathsf{Wake}, \mathsf{sid}_{\mathsf{clock}}, \perp)$ to $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$. Otherwise: $\mathcal{S}$ send $\mathsf{GetTime}$ to $\mathcal{G}_{\text{refClock}}$ to retrieve the current reference time T. Then, $\mathcal{S}$ computes the shift $\sigma_{\text{Sim}} = (T_C - T)$ and sends $(\mathsf{Wake}, \mathsf{sid}_{\mathsf{clock}}, \sigma_{\text{Sim}})$ to $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$.

6.4.2 Analyzing the Accuracy of the Simulator

It is straightforward to verify that the simulator's Wake responses perfectly emulate those in the real world: its simulation of $\mathsf{Continue}$ ensures that time-out actions occur identically in the real and ideal worlds, and otherwise its calculation of σ_{Sim} within the Wake message agrees with the message sent by $\mathcal{A}$.

Therefore, it only remains to show that the answer $\mathcal{S}$ returns can meet the approximate correctness bound required by $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$. If S is corrupted or if C times out then there is no bound to meet. Ergo, in the rest of this section we assume that S is uncorrupted and also that $\delta < \Delta_C$ in response to all queries C makes to $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$ so that no timeout occurs.

In the emulated protocol $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$, the output time T_C is computed by the client as $T_C = T_3 + \frac{1}{2}(\delta - T_3 + T_2)$. T_2 and T_3 are the times returned from the server and are equal to $\mathsf{G}_2 + \sigma_2$ and $\mathsf{G}_3 + \sigma_3$ respectively. δ is returned from $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$ to the client and is computed as $\mathsf{G}_4 + \sigma_4 - \mathsf{G}_1 - \sigma_1$. So, for the emulated client, $T_C = \mathsf{G}_3 + \sigma_3 + \frac{1}{2}(\mathsf{G}_4 + \sigma_4 - \mathsf{G}_1 - \sigma_1 - \mathsf{G}_3 - \sigma_3 + \mathsf{G}_2 + \sigma_2)$.

In the ideal world, the client interacting with $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$ outputs the time $\mathsf{G}_4 + \sigma_{\text{Sim}}$, where the simulator provides σ_{Sim} to account for discrepancy between the emulated client's output and the output of the client interacting with the $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$. Combining the two equations yields:

$$\sigma_{\text{Sim}} = \frac{\mathsf{G}_2 - \mathsf{G}_1 + \mathsf{G}_3 - \mathsf{G}_4}{2} + \frac{\sigma_2 - \sigma_1 + \sigma_3 + \sigma_4}{2}.$$

The simulator must be able to correct for the maximum possible value of σ_{Sim}. It

is straightforward that $|\sigma_{\text{Sim}}|$ is maximized when the following criteria hold.

1. The server's clock is maximally shifted: σ_3 and σ_2 both equal Σ_S.

2. The client's timer shifts to the maximum extent permissible between the Start and TimeElapsed queries: $\sigma_4 = \Sigma_C$ and $\sigma_1 = -\Sigma_C$.

3. The network latency is maximally asymmetric: $\mathcal{E}$ maximizes $(\mathsf{G}_2 - \mathsf{G}_1 + \mathsf{G}_3 - \mathsf{G}_4)$ by incrementing the reference time a large amount between G_1 and G_2 and not at all between G_3 and G_4, or vice versa.

We desire an upper bound on the network asymmetry described in item 3. For the client to avoid timing out, it must be the case that the total time elapsed obey the constraint that

$$(\mathsf{G}_4 + \sigma_4) - (\mathsf{G}_1 + \sigma_1) \leq \Delta_C.$$

It follows that $\mathsf{G}_4 - \mathsf{G}_1 \leq \Delta_C + 2 \cdot \Sigma_C$. Also, since the four times are monotonic, $0 \leq \mathsf{G}_4 - \mathsf{G}_3 \leq \mathsf{G}_4 - \mathsf{G}_1$ and $0 \leq \mathsf{G}_2 - \mathsf{G}_1 \leq \mathsf{G}_4 - \mathsf{G}_1$. Therefore: $|\mathsf{G}_2 - \mathsf{G}_1 + \mathsf{G}_3 - \mathsf{G}_4| \leq \Delta_C + 2 \cdot \Sigma_C$.

Combining the three bounds above yields

$$|\sigma_{\text{Sim}}| \leq \frac{\Delta_C}{2} + \Sigma_C + \Sigma_S.$$

Hence, it suffices for the Σ_C^* for the $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$ to be $\frac{1}{2}\Delta_C + \Sigma_C$ larger than the shift in the server-owned clock $\mathcal{G}_{\text{clock}}^{S,S,\Sigma_S}$ in order for the simulator to be able to simulate correctly, proving Theorem 4.

6.5 More Robust Network Time

In this section, we provide a more robust method to acquire time over the Internet. It is better representative of the way NTP operates: each client queries multiple servers

Protocol $\pi^{\Delta_C,\Sigma_C,\Sigma_S}_{\text{multiTimeSync}}$ [$\mathcal{G}^{C,S_1,\Sigma^*_C}_{\text{clock}}$, ..., $\mathcal{G}^{C,S_n,\Sigma^*_C}_{\text{clock}}$ and $2n$ instances of $\mathcal{F}^{C,\Delta_C,\Sigma_C}_{\text{timer}}$]

GetTime: Begins when the caller sends to C the input $(\mathsf{GetTime}, \mathsf{sid}_{\mathsf{mts}})$. In response, C provisions a timing measurement array $\mathcal{T}_{\mathsf{sid}_{\mathsf{mts}}}$ of length n with all values initialized to a special '?' symbol. C also allocates sufficient storage space to record the session ids of all timers and clocks with which it interacts. Finally, C invokes QueryClock.

QueryClock: Begins when invoked by GetTime or Continue. Invariant: at least one clock has not been invoked.

1. C identifies a previously-unqueried $\mathcal{G}^{C,S_i,\Sigma^*_C}_{\text{clock}}$.
2. C sends a Start command to the i^{th} timer and waits for an ok response.
3. C sends the command $(\mathsf{GetTime}, \mathsf{sid}_{\mathsf{clock}i})$ to $\mathcal{G}^{C,S_i,\Sigma^*_C}_{\text{clock}}$.

ResponseReceived: Begins when C receives a response $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{clock}i}, T_i)$ from a clock $\mathcal{G}^{C,S_i,\Sigma^*_C}_{\text{clock}}$:

1. C records $\mathcal{T}_{\mathsf{sid}_{\mathsf{mts}}}[i] \leftarrow T_i$.
2. C sends a TimeElapsed message to the i^{th} timer. If it returns $\perp$, then C updates $\mathcal{T}_{\mathsf{sid}_{\mathsf{mts}}}[i] \leftarrow \perp$.
3. If $\mathcal{T}_{\mathsf{sid}_{\mathsf{mts}}}[i] \neq \perp$, then C sends a Start command to the $(n+i)^{\text{th}}$ timer and waits for an ok response.
4. Invoke the Continue routine.

Continue: Begins when $\mathcal{A}$ sends to C the Continue command or when ResponseReceived ends.

1. If C has not yet queried each $\mathcal{G}^{C,S_i,\Sigma^*_C}_{\text{clock}}$, then C begins the QueryClock protocol as stated above.
2. Else, C begins the CheckTimeout protocol as stated below.

CheckTimeout: Begins when invoked by Continue. Invariant: each $\mathcal{G}_{\text{clock}}^{C,S_i,\Sigma_C^*}$ has already received a GetTime query.

1. For all i: C sends a TimeElapsed message to the i^{th} timer if $\mathcal{T}_{\mathsf{sid}_{\mathsf{mts}}}[i] == ?$ and to the $(n+i)^{\text{th}}$ timer otherwise. If the timer returns a $\perp$ response, then C updates $\mathcal{T}_{\mathsf{sid}_{\mathsf{mts}}}[i] \leftarrow \perp$.
2. If none of the records in $\mathcal{T}_{\mathsf{sid}_{\mathsf{mts}}}$ equal '?', then invoke Finalize. Else, end the current activation.

Finalize: Begins when invoked by CheckTimeout. Note that $\mathcal{A}$ never gets control during Finalize.

1. For all i such that $\mathcal{T}_{\mathsf{sid}_{\mathsf{mts}}}[i] \neq \perp$, send a TimeElapsed message to the $(n+i)^{\text{th}}$ timer and wait for a response of the form $(\delta, \mathsf{sid}_{\mathsf{timer}n+i})$.
   - If $\delta == \perp$, then update $\mathcal{T}_{\mathsf{sid}_{\mathsf{mts}}}[i] \leftarrow \perp$.
   - Otherwise, update $\mathcal{T}_{\mathsf{sid}_{\mathsf{mts}}}[i] \leftarrow \mathcal{T}_{\mathsf{sid}_{\mathsf{mts}}}[i] + \delta$.
2. C sets T_C to be the median of the non-$\perp$ values within $\mathcal{T}_{\mathsf{sid}_{\mathsf{mts}}}$. If all values equal $\perp$, then C sets T_C to $\perp$.
3. Output $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{mts}}, T_C)$ to the caller.

Figure 6·9: Time Synchronization Protocol $\pi_{\text{multiTimeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ in between a client C with access to unused $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$ functionalities and a set of n $\mathcal{G}_{\text{clock}}^{C,S_i,\Sigma_C^*}$ functionalities each parameterized by a server S_i from the set $\{S_1, ..., S_n\}$.

to increase its resilience to compromise, and different clients query different servers to remove network and resource bottlenecks.

Section 6.5.1 considers the case of a single client accessing multiple servers. Then, Section 6.5.2 considers the multi-stratum case in which each server in a stratum receives its notion of time not from a local clock, but instead by acting also as a client and querying several servers in the stratum below. We use the composition theorem to provide modular and relatively simple analysis of these these rather intricate interactions.

Functionality $\mathcal{G}_{\text{multiClock}}^{P,\mathcal{S},\Sigma}$ [$\mathcal{G}_{\text{refClock}}$]

This ideal functionality is identified by a session id $\mathsf{sid}_{\mathsf{mclock}} = (\mathsf{sid}'_{\mathsf{mclock}}, P, \{S_1, ..., S_n\})$ where $\mathsf{sid}'_{\mathsf{mclock}} = \mathsf{sid}_{\mathsf{clock}1} \ldots \mathsf{sid}_{\mathsf{clock}n}$ that denotes the clock's owner P as well as a set of parties $\mathcal{S} = \{S_1, ..., S_n\}$ whose honesty influences the accuracy of the clock. It is also parameterized by the maximum allowable shift Σ from the reference time. It operates as follows.

Corrupt: Upon receiving a message $(\mathsf{Corrupt}, S)$, if $S \in \{S_1, \ldots, S_n\}$ then record S_i as corrupted.

GetTime: Upon receiving input $(\mathsf{GetTime}, \mathsf{sid}_{\mathsf{mclock}})$ from party P', ignore this request if $P' \neq P$, otherwise:

1. Send $(\mathsf{Sleep}, \mathsf{sid}_{\mathsf{mclock}})$ to the adversary. Wait for a response from the adversary, of the form $(\mathsf{Wake}, \mathsf{sid}_{\mathsf{mclock}}, \sigma, L)$ where L is the list of servers that are deemed to provide non-$\perp$ timing measurements.
2. If $\sigma == \perp$, output $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{clock}}, \perp)$ to P.
3. Else send $\mathsf{GetTime}$ to $\mathcal{G}_{\text{refClock}}$. Denote its response as T.
   (a) If the majority of servers in L are corrupted or $|\sigma| \leq \Sigma$, output $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{mclock}}, T + \sigma)$ to P.
   (b) Else output $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{mclock}}, \perp)$ to P.

Figure 6·10: Ideal functionality $\mathcal{G}_{\text{multiClock}}^{P,\mathcal{S},\Sigma}$ that outputs a time influenced by the corruption status of servers in $\mathcal{S}$. Note that the $\mathcal{G}_{\text{clock}}^{P,S,\Sigma}$ functionality in Fig. 6·3 is a special case of this one with a singleton set $\mathcal{S} = \{S\}$.

6.5.1 Multiple Server Time Sync

At a high level, the new $\pi_{\text{multiTimeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ protocol involves a client who queries n different servers for the time. Once all timing measurements have been collected or time-out, then the client outputs the *median* of all non-$\perp$ timing measurements.

The full protocol to aggregate times from multiple servers is shown in Fig. 6·9. This protocol requires the client to keep an extra timer per server in order to calculate and remember the freshness of responses.

Producing a real multi-server protocol should involve the composition of $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ protocols with each server. Thanks to the UC composition theorem, it suffices to analyze a simpler protocol in which the $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ with each server S_i is replaced with its corresponding ideal functionality $\mathcal{G}_{\text{clock}}^{C,S_i,\Sigma_C^*}$. We make a few remarks about this use of composition:

- There are implicitly two uses of $\mathcal{G}_{\text{clock}}$ here: the server's clock $\mathcal{G}_{\text{clock}}^{S,S,\Sigma_S}$ in the real protocol and the entire ideal functionality $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$. We stress the lack of circularity here, as shown in Fig. 6·1: the first clock is a subroutine of $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ whereas the second clock is an ideal abstraction of it.

- The protocol $\pi_{\text{multiTimeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ contains Σ_S as a parameter but its specification in Fig. 6·9 never mentions Σ_S explicitly. Instead, the only impact of Σ_S is its influence over Σ_C^*, as shown in Theorem 4.

In comparison to the single-server case, this protocol offers one drawback and one benefit. The extra timer adds the price of $2 \cdot \Sigma_C$ additional shift to the time computed by the client. On the plus side, the multiple server protocol can guarantee approximate correctness even if some servers are corrupted, due to the following two observations. First, uncorrupted measurements must be close to the reference time G. Second, if the majority of servers are uncorrupted, then the median time must be bounded on both sides by uncorrupted samples.

These observations yield an approximate correctness guarantee that is quite robust! We do *not* require that all or even most timing measurements reach the client; on the contrary, the adversary might corrupt, drop, or delay almost all requests. Additionally, the adversary may corrupt parties adaptively and may choose their responses conditioned upon the timing measurements of the honest parties. We simply require the following constraint: of the servers whose interactions result in the client

receiving a timing measurement (i.e., anything but $\perp$), a majority of those servers are uncorrupted.[1]

The ideal functionality $\mathcal{G}_{\text{multiClock}}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$ specified in Figure 6·10 formally captures this accuracy constraint.

Theorem 5 (Multiple server UC security)**.** *Given parameters that satisfy* $\Sigma'_C \geq 2.5 \cdot \Delta_C + 3 \cdot \Sigma_C + \Sigma_S$*, the multiple server approximate time synchronization protocol* $\pi_{multiTimeSync}^{\Delta_C,\Sigma_C,\Sigma_S}$ *GUC-realizes* $\mathcal{G}_{multiClock}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$*.*

As before, the simulator $\mathcal{S}$ internally runs an emulated copy of the real world protocol $\pi_{\text{multiTimeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$, where $\mathcal{S}$ plays the role of $\mathcal{E}$ inside this simulation. In particular, $\mathcal{S}$ internally emulates the execution of adversary $\mathcal{A}$, all n $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$ functionalities, all the instances of $\mathcal{F}_{\text{auth}}$, and each of the involved parties. In addition, $\mathcal{S}$ relays the messages sent from the emulated parties to $\mathcal{G}_{\text{refClock}}$ and $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$, and from $\mathcal{G}_{\text{refClock}}$ and $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$ to the emulated parties.

While conducting this simulation, $\mathcal{S}$ monitors the traffic of its emulated $\mathcal{A}$ and $\mathcal{E}$, along with any messages that $\mathcal{S}$ directly receives from $\mathcal{G}_{\text{multiClock}}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$. The messages that $\mathcal{S}$ views/receives causes it to make changes in the ideal world or the emulated world.

Simulating GetTime**:** When $\mathcal{G}_{\text{multiClock}}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$ sends a message of the form $(\mathsf{Sleep}, \mathsf{sid}_{\mathsf{mclock}})$, then $\mathcal{S}$ instantiates C with the environmentally-provided message $(\mathsf{GetTime}, \mathsf{sid}_{\mathsf{mts}})$.

Simulating Continue**:** When $\mathcal{S}$ observes $\mathcal{A}$ sending a Continue message to the emulated C, then $\mathcal{S}$ sends a Continue message to C. (Upon receiving this message, the emulated C might invoke QueryClock or CheckTimeout.)

[1]This threshold constraint corresponds to the strong "sleepy model of consensus" of (Pass and Shi, 2016) and (Micali, 2016).

Maintaining Records: Upon receiving a message of the form $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{clock}i}, T_i)$ from $\mathcal{G}_{\text{clock}}^{C,S_i,\Sigma_C^*}$, if $T_i == \perp$, initialize an empty list L if there does not exist one. Append i to list L. Subsequently, if a $\mathsf{CheckTimeout}$ procedure ever returns a $\perp$ when querying the i^{th} timer, then remove i from L.

Relaying messages to $\mathcal{E}$: When the emulated $\mathcal{A}$ sends a message m to the emulated $\mathcal{E}$, then $\mathcal{S}$ relays m to the real environment. Conversely, $\mathcal{S}$ forwards messages sent by the real $\mathcal{E}$ to the emulated $\mathcal{A}$.

Corrupting a server: When $\mathcal{A}$ sends a $(\mathsf{Corrupt}, S_i)$ message in the emulated world, then $\mathcal{S}$ sends a $(\mathsf{Corrupt}, S_i)$ message to $\mathcal{G}_{\text{multiClock}}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$.

Completing the simulation: When the emulated C sends to $\mathcal{E}$ its output $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{mts}}, T_C)$, then $\mathcal{S}$ knows both *when* and *what* to send back to $\mathcal{G}_{\text{multiClock}}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$. If $T_C == \perp$, then $\mathcal{S}$ sends $(\mathsf{Wake}, \mathsf{sid}_{\mathsf{mclock}}, \perp, \perp)$ to $\mathcal{G}_{\text{multiClock}}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$. Otherwise: $\mathcal{S}$ sends $\mathsf{GetTime}$ to $\mathcal{G}_{\text{refClock}}$ to retrieve the current reference time T and also retrieves the list L. Then, $\mathcal{S}$ computes the shift $\sigma_{sim} = (T_C - T)$ and sends $(\mathsf{Wake}, \mathsf{sid}_{\mathsf{mclock}}, \sigma_{sim}, L)$ to $\mathcal{G}_{\text{multiClock}}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$.

In the ideal model (namely in the execution of $\mathcal{G}_{\text{multiClock}}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$ with $\mathcal{S}$), $\mathcal{G}_{\text{multiClock}}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$ receives from $\mathcal{S}$ an offset σ'_{Sim} and a list of the servers whose $\mathcal{G}_{\text{clock}}^{P,S,\Sigma}$ boxes output $\perp$ in the simulated $\pi_{\text{multiTimeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$. It is also informed when a server is corrupted.

From this information $\mathcal{G}_{\text{multiClock}}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$ computes T_C as $\mathsf{G}_x + \sigma'_{\text{Sim}}$ where, if fewer than half of the servers whose $\mathcal{G}_{\text{clock}}^{S,S,\Sigma_S}$ boxes did not output $\perp$ are corrupted, $\sigma'_{\text{Sim}} \leq \Sigma'_C$, the maximum offset that a $\mathcal{G}_{\text{multiClock}}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$ box will allow.

The value output by $\pi_{\text{multiTimeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ is selected as the median of the values returned from its $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$ boxes plus a δ corresponding to the time passed since the response was received. The value returned from $\mathcal{G}_{\text{clock}}^{C,S_i,\Sigma_C^*}$ is of the form $T_i = \mathsf{G}_i + \sigma_i$ where

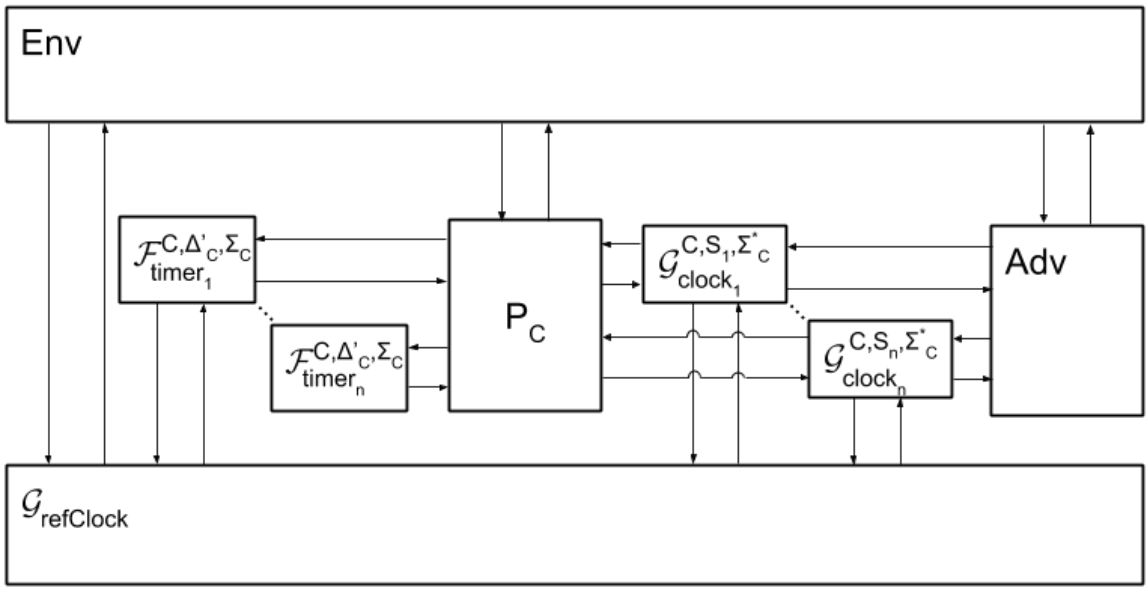

Figure 6·11: Interactions between participants in the real world execution of multi-server time sync

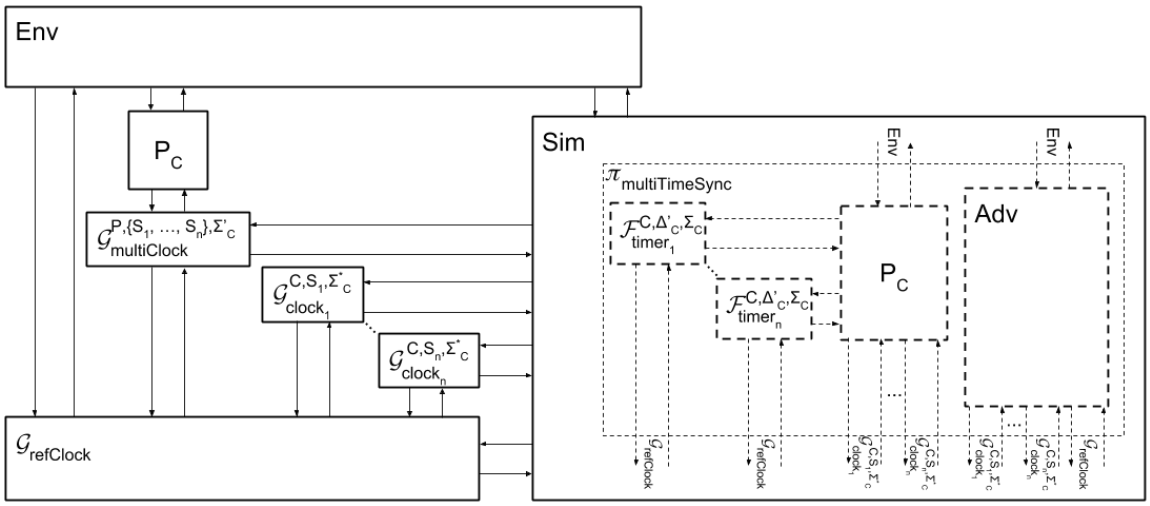

Figure 6·12: Interactions between participants in the ideal world execution of multi-server time sync, including the emulation of the real world inside of the simulator.

$|\sigma_i| \leq \Sigma^*_{C_{S_i}}$ the max offset for the $\mathcal{G}^{C,S,\Sigma^*_C}_{\text{clock}}$ owned by server S_i if S_i is uncorrupted. The client in $\pi^{\Delta_C,\Sigma_C,\Sigma_S}_{\text{multiTimeSync}}$ will then add a δ to this corresponding to the elapsed time since receiving the response T_i from the i^{th} $\mathcal{G}^{C,S,\Sigma^*_C}_{\text{clock}}$. The delay δ is computed like before as $\mathsf{G}_2 + \sigma_2 - \mathsf{G}_1 - \sigma_1$ and is at most Δ'_C. Finally, it must be the case that $\Delta'_C > \Delta_C$ or else valid responses could timeout while C is waiting for all $\mathcal{G}^{C,S_i,\Sigma^*_C}_{\text{clock}}$ to respond.

In this case $\mathsf{G}_2 = \mathsf{G}_x$ as the $\mathcal{F}^{C,\Delta_C,\Sigma_C}_{\text{timer}}$ is queried for a δ when all the responses are received in the simulation and there is not a chance for the environment to update the reference time between this point and when it is obtained by $\mathcal{G}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}_{\text{multiClock}}$. Additionally, $\mathsf{G}_1 = \mathsf{G}_i$ as the timer is started once the response is received. Therefore, $\mathsf{G}_x - \mathsf{G}_i \leq \Delta'_C + 2 * \Sigma_C$ where Σ_C is, as before, the max allowable shift for a C's $\mathcal{F}^{C,\Delta_C,\Sigma_C}_{\text{timer}}$.

The T_C output by $\pi^{\Delta_C,\Sigma_C,\Sigma_S}_{\text{multiTimeSync}}$ is $\mathsf{G}_i + \sigma_i + \delta_i$ so in order to properly simulate $\mathcal{S}$ must be able to input a σ_{Sim} that will make $\mathsf{G}_x + \sigma'_{\text{Sim}}$ equal to $\mathsf{G}_i + \sigma_i + \delta$. G_x is at most $\Delta'_C + 2 \cdot \Sigma_C$ greater than G_i, σ_i is at most $\Sigma^*_{C_{S_i}}$, and $\delta \leq \Delta'_C$. Finally, recall from Section 6.4.2 that $\Sigma^*_{C_{S_i}} \leq \frac{\Delta_C}{2} + \Sigma_C + \Sigma_{S_i}$.

Combining the above bounds yields

$$|\sigma'_{\text{Sim}}| \leq 2.5 \cdot \Delta_C + 3 \cdot \Sigma_C + \Sigma_S.$$

Therefore, it suffices for the Σ'_C for the $\mathcal{G}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}_{\text{multiClock}}$ to be $2(\Delta'_C + \Sigma_C)$ larger than the Σ^*_C for the simulated $\mathcal{G}^{C,S,\Sigma^*_C}_{\text{clock}}$ in order for the simulator to be able to simulate correctly, proving Theorem 5.

6.5.2 Multiple Strata Network Time

In this section, we add another feature of the network time protocol: a hierarchical structure to distribute the network load required to propagate network time. Partici-

pants in NTP are stratified, with stratum-0 servers possessing their own source of time $\mathcal{G}_{\text{clock}}^{S,S,\Sigma_0}$ and all other participants serving as both clients *and* servers. We restrict our attention to the case in which NTP servers communicate within the following 'rigid topology.'

- We insist that each individual machine be statically pegged to a single stratum forever. We impose this restriction so that we may compose invocations of $\pi_{\text{multiTimeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$. In the more realistic scenario where machines can change strata, the UC composition theorem breaks down since a feedback loop may occur where a client provides an input into its own time measurement.

- We require that machines in stratum j only take timing measurements from servers located in stratum $j-1$ and thus only provide time to stratum $j+1$ clients. We impose this restriction merely to simplify our calculations in Theorem 6; the UC composition theorem would enable more complicated analyses if so desired.

- We use the following parameters: Stratum 0 servers are within shift Σ_0 of $\mathcal{G}_{\text{refClock}}$. All machines in higher strata have timers with maximum delay Δ^* and maximum shift Σ^*.

Additionally, we make two observations that generalize the work we have already provided. First $\pi_{\text{multiTimeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ continues to be well-defined if it receives multi-server ideal functionalities $\mathcal{G}_{\text{multiClock}}^{P,\mathcal{S},\Sigma}$ as subroutines rather than single-server functionalities. Additionally, if all servers within all of the $\mathcal{G}_{\text{multiClock}}^{P,\mathcal{S},\Sigma}$ used by the client have the same Σ_S bound, then the statement and proof of Theorem 5 continue to hold in this setting.

Second, we may further generalize $\pi_{\text{multiTimeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ by permitting $\mathcal{S}$ to be a set of sets and by modifying the consistency rule in step 3a to state that the adversary

only has free reign to alter the time if the set of servers $\mathcal{S}$ is 'bad,' as defined below. Furthermore, Theorem 5 continues to provide bounds on consensus and accuracy in this case as well.

Definition 3. During an execution of $\pi^{\Delta_C,\Sigma_C,\Sigma_S}_{\text{multiTimeSync}}$, we denote a server as *bad* if it is corrupted. Additionally, a set of parties is deemed to be *bad* if a majority of elements are corrupted. Note that these elements may either be parties or sets themselves; in the latter case, the notion of corruptedness is defined recursively.

Here, the majority vote is only taken over elements that respond to the client's request for timing measurements within its maximum allowable delay Δ (which we stress that the adversary has the capacity to control). The fact that non-responsive servers do not factor either positively or negatively into the badness of a set of parties is consistent with the sleepy model of consensus (cf. footnote 1).

These two observations and the UC composition theorem allow us to bound the worst-case error when timing measurements percolate down multiple strata.

Theorem 6. *Consider several machines who conduct multiple server timing measurements following the network topology specified above. Then, a client C at stratum j who executes $\pi^{\Delta_C,\Sigma_C,\Sigma_S}_{multiTimeSync}$ using the set of servers $\mathcal{S}$ will receive a time whose inaccuracy is bounded by $\Sigma_j \geq 2.5j \cdot \Delta^* + 3j \cdot \Sigma^* + \Sigma_0$ as long as the set $\mathcal{S}$ is not bad.*

The proof of this theorem is straightforward. First, we apply the UC theorem to replace all instances of the $\pi^{\Delta_C,\Sigma_C,\Sigma_S}_{\text{multiTimeSync}}$ protocol (for C and for all of the timeservers who get their measurements from lower strata as well) with instances of the ideal functionality $\mathcal{G}^{P,\mathcal{S},\Sigma}_{\text{multiClock}}$. We remark that our definition of good and bad timeservers matches precisely with the (modified) constraint for timing consensus in step 3a of $\mathcal{G}^{P,\mathcal{S},\Sigma}_{\text{multiClock}}$. Ergo, Theorem 5 upper bounds the inaccuracy of all good servers at stratum i as $\Sigma_i \geq 2.5\Delta^* + 3\Sigma^* + \Sigma_{i-1}$. Summing these inequalities for all $i \in \{1, 2, \ldots, j\}$ yields the desired result.

We stress that Theorem 6 provides a worst case bound. By contrast, in the remainder of this section we mathematically analyze and computationally simulate average

case error propagation over multiple strata by timeservers with *accidental* rather than adversarial timing inaccuracies. Suppose that these accidental network asymmetries and server clock imprecisions contribute to delays δ and shifts σ (respectively) that are randomly distributed (e.g., using a uniform or Gaussian distribution). In this average case setting, the central limit theorem provides much more stringent bounds on error propagation.

- Within a single stratum, the fact that each client invokes multiple servers means that their individual shift errors are very likely to interfere destructively. Network jitter effects do cause a noticeable delay, however.

- The effect of network asymmetry at a particular stratum i upon the shift (as found in Theorems 4 and 5) is also reduced significantly when progressing down the strata.

The net result of the average case analysis is that the expected shift at a high stratum is influenced mostly by the magnitude of the network asymmetry at that stratum only. Hence, a high stratum timeserver with low latency network connections may actually possess *better* timing measurements than a low stratum timeserver in a high latency environment, even if the latter contributes toward the timing measurements of the former.

6.6 Using Approximate Time in UC Protocols

In this section, we explore the ramifications of injecting approximate time into existing, time-agnostic GUC protocols and functionalities. Although our principal interest is in the expiration and revocation of PKI certificates, much of our analysis applies generically to any protocol whose security depends in part on the approximate accuracy of time.

6.6.1 Adding Time to Existing Protocols & Functionalities

We begin by considering generically the influence of time upon existing UC protocols and functionalities that have previously been proved secure in the usual untimed, asynchronous setting. The following straightforward theorem states that UC security continues to hold for all untimed protocols:

Theorem 7. *Let π be a protocol that GUC-realizes functionality $\mathcal{G}$ in an untimed setting. Then, π continues to GUC-realize $\mathcal{G}$ even in the presence of exact or approximate time functionalities like $\mathcal{G}_{refClock}$ or $\mathcal{G}_{clock}^{P,S,\Sigma}$.*

This theorem follows immediately from the UC security guarantee in the presence of global functionalities. Since no environment can distinguish π from $\mathcal{G}$, in particular this condition must hold for environments that either keep track of, or have access to, a time functionality such as G.

More interestingly, we can automatically *add* a time dependency on top of protocols that previously lacked an understanding of time. In this section, we focus upon protocols and functionalities of the following type.

Definition 4 (Binary decider)**.** We say that a protocol π_{bin} is a *binary decider* if the following constraints hold:

- Only one party P receives output. We denote the collection of inputs by $\vec{x}$ and the output as (b, y).
- The value b is a single bit. (By contrast, y and the elements of $\vec{x}$ are strings of arbitrary length.)

Intuitively, binary deciders provide P with a putative output and a *verification bit* that determines whether P chooses to accept the answer. Protocols of this form include bit commitments, zero-knowledge proofs, and (of particular interest to us) signature and certificate verification checks.

Given any binary decider protocol π_{bin}, Fig. 6·13 constructs a new protocol $\hat{\pi}_{\text{bin}}^{\Sigma,t^*}$ that operates identically to π_{bin} except that it subjects the verification bit to a new

constraint that rejects responses when P's time is past a threshold t^*. For simplicity, in this section we assume that P's clock $\mathcal{G}_{\text{clock}}^{P,P,\Sigma}$ is corrupted only if P is, so the only relevant parameter is the maximum clock shift Σ. Note that t^* and Σ are explicitly provided to the adversary; $\hat{\pi}_{\text{bin}}^{\Sigma,t^*}$ makes no attempt to hide them.

Next, we demonstrate a canonical method to transform ideal functionalities analogously. Given any $\mathcal{G}_{\text{bin}}$, Fig. 6·14 constructs a new ideal functionality $\hat{\mathcal{G}}_{\text{bin}}^{\Sigma,t^*}$ that executes $\mathcal{G}_{\text{bin}}$ as a subroutine and also determines the reference time G. It is more restrained than before: the adversary can only change an otherwise-valid response if G is close to the threshold t^*.

Next, we show that these two transformations produce identical outcomes. Intuitively, the adversary can control the output bit of $\hat{\mathcal{G}}_{\text{bin}}^{\Sigma,t^*}$ only when her clock skew capability can be used to affect $\hat{\pi}_{\text{bin}}^{\Sigma,t^*}$'s output.

Theorem 8. *Suppose that the binary decider π_{bin} GUC-realizes $\mathcal{G}_{bin}$ and that π_{bin}'s output party P has access to a clock $\mathcal{G}_{clock}^{P,P,\Sigma}$. Then, $\hat{\pi}_{bin}^{\Sigma,t^*}$ GUC-realizes $\hat{\mathcal{G}}_{bin}^{\Sigma,t^*}$.*

Let $\mathcal{D}_\pi$ denote the dummy adversary against π_{bin} and $\mathsf{Sim}_{\mathcal{G}}$ denote its corresponding simulator. Additionally, let $\mathcal{D}_{\hat{\pi}}$ denote the dummy adversary against $\hat{\pi}_{\text{bin}}^{\Sigma,t^*}$. Our objective is to design a simulator $\mathsf{Sim}_{\hat{\mathcal{G}}}$ that connects with $\hat{\mathcal{G}}_{\text{bin}}^{\Sigma,t^*}$ and produces a view indistinguishable from $\mathcal{D}_{\hat{\pi}}$.

$\mathsf{Sim}_{\hat{\mathcal{G}}}$ emulates the real world interaction with $\mathcal{D}_{\hat{\pi}}$ in its head, and it behaves as follows during each step of the execution of $\hat{\mathcal{G}}_{\text{bin}}^{\Sigma,t^*}$ in the ideal world.

1. $\mathsf{Sim}_{\hat{\mathcal{G}}}$ sends (t^*, Σ) to $\mathcal{E}$ and relays $\mathcal{E}$'s response to $\hat{\mathcal{G}}_{\text{bin}}^{\Sigma,t^*}$, just as $\mathcal{D}_{\hat{\pi}}$ does with $\hat{\pi}_{\text{bin}}^{\Sigma,t^*}$.

2. During the execution of $\mathcal{G}_{\text{bin}}$, the simulator $\mathsf{Sim}_{\hat{\mathcal{G}}}$ simply acts as $\mathsf{Sim}_{\mathcal{G}}$ would.

3. $\mathsf{Sim}_{\hat{\mathcal{G}}}$ observes the shift σ that $\mathcal{D}_{\hat{\pi}}$ applies to P's clock and then sends $b' = b \wedge [t \overset{?}{\leq} t^*]$ to $\hat{\mathcal{G}}_{\text{bin}}^{\Sigma,t^*}$.

Protocol $\hat{\pi}_{\text{bin}}^{\Sigma,t^*}$ [π_{bin}, $\mathcal{G}_{\text{clock}}^{P,P,\Sigma}$]

When instantiated with inputs $\vec{x}$ where party P's input has the form $x_P = (x'_P, t^*, \Sigma)$, do the following:

1. P sends (t^*, Σ) to $\mathcal{A}$ and waits for an ok response.
2. The parties execute the subroutine protocol π_{bin} on inputs $\vec{x}'$, where $x'_{P'} = x_{P'}$ for all $P' \neq P$. Eventually, P produces output of the form (b, y).
3. Before submitting this output, P queries her clock (with max shift Σ) for the current time t. This request invokes $\mathcal{A}$ to provide a shift σ.
4. Compute $b' = b \wedge [t \overset{?}{\leq} t^*]$.
5. P outputs (b', y) to the caller.

Figure 6·13: Time-conditional protocol $\hat{\pi}_{\text{bin}}^{\Sigma,t^*}$. It connects to two subroutines: a binary decider π_{bin} and P's clock.

Note that $\mathsf{Sim}_{\hat{\mathcal{G}}}$ has nothing to do during steps 4-5.

The messages sent to the environment during steps 1-2 are clearly identical to those of $\mathcal{D}_{\hat{\pi}}$. The only other message received by $\mathcal{E}$ is the output (b', y).

Ergo, to prove simulation, it suffices to show that the output values b' are identical in $\hat{\pi}_{\text{bin}}^{\Sigma,t^*}$ and $\hat{\mathcal{G}}_{\text{bin}}^{\Sigma,t^*}$. This follows from the fact that the adversary's ability to shift P's clock is bounded such that:

$$[t \overset{?}{\leq} t^*] = \begin{cases} 1, \text{ if } \mathsf{G} < t^* - \Sigma, \\ 0, \text{ if } \mathsf{G} > t^* + \Sigma, \\ \text{controlled by } \mathsf{Sim}_{\hat{\mathcal{G}}}, \text{ otherwise.} \end{cases}$$

Hence, $\mathsf{Sim}_{\hat{\mathcal{G}}}$'s inability to influence b' in the first two cases of step 4 is irrelevant because $\hat{\pi}_{\text{bin}}^{\Sigma,t^*}$'s output must equal b and 0, respectively. In the third case, $\mathsf{Sim}_{\hat{\mathcal{G}}}$ chooses b' just as $\hat{\pi}_{\text{bin}}^{\Sigma,t^*}$ does during step 4, so the simulation is perfect.

Functionality $\hat{\mathcal{G}}_{\text{bin}}^{\Sigma,t^*}$ [$\mathcal{G}_{\text{bin}}$, $\mathcal{G}_{\text{refClock}}$]

When instantiated with inputs $\vec{x}$ where party P's input has the form $x_P = (x'_P, t^*, \Sigma)$, do the following:

1. Send (t^*, Σ) to $\mathcal{A}$. Wait for an ok response.
2. Send $\vec{x}'$ (as defined in Fig. 6·13) to subroutine $\mathcal{G}_{\text{bin}}$. Eventually, receive a response of the form (b, y).
3. Query the adversary for a value b'.
4. Obtain the reference time G from $\mathcal{G}_{\text{refClock}}$. Update the value of b' as follows:
   - If $\mathsf{G} < t^* - \Sigma$, then set $b' = b$.
   - If $\mathsf{G} > t^* + \Sigma$, then set $b' = 0$.
   - If $b = 0$, set $b' = 0$.
5. Output (b', y) to P.

Figure 6·14: The time-conditional $\hat{\mathcal{G}}_{\text{bin}}^{\Sigma,t^*}$ functionality with two subroutines: $\mathcal{G}_{\text{refClock}}$ and an untimed binary decider $\mathcal{G}_{\text{bin}}$. The max shift Σ of P's clock affects the behavior of $\hat{\mathcal{G}}_{\text{bin}}^{\Sigma,t^*}$, even though they never communicate.

Functionality $\mathcal{G}_{\text{timed-bb}}$

Report: Upon receiving from party P a message of the form $(\mathsf{Register}, P, \mathsf{serial}, v, t)$, send the message to the adversary and wait for an ok response. If this is the first request involving (P, serial) then record the tuple $(P, \mathsf{serial}, v, t)$. Otherwise, ignore the message.

Retrieve: Upon receiving from some party P_i or the adversary a message $(\mathsf{Retrieve}, P_j, \mathsf{serial})$, retrieve the record r containing (P_j, serial) and return $(\mathsf{Retrieve}, r)$. If no such record exists, return $(\mathsf{Retrieve}, \perp)$.

ChangeExpiration: Upon receiving from party P a message of the form $(\mathsf{ChangeExpiration}, P, \mathsf{serial}, t')$, retrieve the record of the form $(P, \mathsf{serial}, v, t)$. If $t' < t$, then replace t with t' in this record. Otherwise (including if the record does not exist), do nothing.

Figure 6·15: A public bulletin board that augments (Canetti et al., 2016, Fig. 3) to incorporate an expiration time.

6.6.2 Application to Public Key Infrastructure

In this section, we augment Canetti, Shahaf, and Vald's GUC analysis of signature-based authentication (Canetti et al., 2016) to enable revocation and expiration. The goal of (Canetti et al., 2016) is to provide, within the UC framework, modeling and analysis of the process of (1) generating signing and verification keys for a digital signature scheme, (2) certifying the verification key, and (3) using these keys to authenticate and verify messages by way of signing them on the sending end and verifying the verification key and the signature on the receiving end. An important innovation within (Canetti et al., 2016) is to provide adequate treatment to the fact that the certified verification keys are universally available and may be used to authenticate several messages within many different protocols. The main components of their modeling are:

- A public bulletin-board $\mathcal{G}_{\text{bb}}$ where parties can publicly associate values with their identities. The bulletin board is globally available and guarantees authenticity (a party can only associate values with her own true ID).

- A certification functionality $\mathcal{G}_{\text{cert}}$ that provides its owner with a public key, that it then posts globally using $\mathcal{G}_{\text{bb}}$. When the owner provides a message m to be signed, $\mathcal{G}_{\text{cert}}$ returns an idealized signature string S; later on, when asked by anyone, $\mathcal{G}_{\text{cert}}$ correctly verifies valid message-signature pairs.

- An existentially unforgeable signature scheme $\mathcal{F}_{\text{sig}}$ that is used to realize $\mathcal{G}_{\text{cert}}$.

- An authenticated message transmission functionality $\mathcal{F}_{\text{cert-auth}}$ that properly models the non-deniability of an authenticated message, i.e., that it is possible for third parties to verify whether a given message was indeed sent and signed by the sender. (This stands in contrast with the "standard" authenticated message

transmission functionality in Fig. 6·2, which only allows the specified receiver to tell whether a message is authentic.)

- A protocol π_{auth} for realizing $\mathcal{F}_{\text{cert-auth}}$ using $\mathcal{G}_{\text{cert}}$, by way of signing the message by the sender and later verifying the signature on the receiving end, against $\mathcal{G}_{\text{cert}}$ of the sender. An important aspect of this analysis is that $\mathcal{G}_{\text{bb}}$ is global and exists regardless of the specific instance of $\mathcal{G}_{\text{cert}}$. Furthermore, while $\mathcal{G}_{\text{cert}}$ is specific for a single "party" (i.e., long-term entity), it is global in the sense that it exists regardless of any single message-authentication instance.

We remark that, while the analysis of (Canetti et al., 2016) only considers the setting in which each party registers a single certificate, one can verify that their modeling and proofs continue to hold when each pid is replaced with a $(\mathsf{pid}, \mathsf{serial})$ pair, where serial denotes a unique identifier of a certificate issued by a particular CA (Cooper et al., 2008, §4.1.2.2). As a result, the same analysis applies when participants can request the creation of multiple certificates, which is essential when certificates expire.

Adding time awareness and certificate revocation. To capture expiration and revocation requests, we extend Canetti et al.'s public bulletin board $\mathcal{G}_{\text{bb}}$ into a time-aware bulletin board $\mathcal{G}_{\text{timed-bb}}$ that supports expiration and revocation. Our extension augments the Report method to record the expiration time and adds a new third method called ChangeExpiration to support revocations. Figure 6·15 shows the details.

Then, we may apply Theorem 8 to "lift" the certification functionality $\mathcal{G}_{\text{cert}}$ and the authentication functionality $\mathcal{F}_{\text{cert-auth}}$ described in (Canetti et al., 2016) to their respective time-dependent versions. In the lifted $\hat{\mathcal{G}}_{\text{cert}}^{\Sigma,t^*}$, the recipient determines the appropriate threshold t^* to use by querying $\mathcal{G}_{\text{timed-bb}}$ with the sender's credentials. Finally, the timed version of the real authentication protocol $\pi_{\text{time-auth}} \triangleq \hat{\pi}_{\text{auth}}^{\Sigma,t^*}$ GUC-

realizes the timed ideal functionality $\mathcal{F}_{\text{time-cert-auth}} \triangleq \hat{\mathcal{F}}^{\Sigma,t^*}_{\text{cert-auth}}$ based upon Theorem 8 and (Canetti et al., 2016, Claim 4.4).

This protocol $\pi_{\text{time-auth}}$ combines all of the components designed so far to provide time-based non-deniable authentication, as shown in Fig. 6·1. It requires a clock, which we know how to instantiate from Sections 6.4-6.5. Additionally, it uses $\mathcal{G}_{\text{cert}}$ as a subroutine just as its untimed counterpart did.

By imbuing this subroutine with a notion of time itself, $\hat{\mathcal{G}}^{\Sigma,t^*}_{\text{cert}}$ can interface with our timed bulletin board $\mathcal{G}_{\text{timed-bb}}$ to attest that (1) the signature is valid, just as before and (2) the certificate hasn't yet expired, using the new expiration time t contained within the record returned by $\mathcal{G}_{\text{timed-bb}}$'s Retrieve command. Furthermore, in case of key compromise, the signer can request that her certificate be revoked.

Impact upon use of the PKI today. An important lesson learned from this modeling is that real-life certificate revocation lists and online certificate status requests must continue to answer requests about revoked or expired certificates during the interval $[t^*, t^* + \Sigma]$ because clients may not be able to adjudicate them correctly on their own before this time. Here, Σ denotes the maximum shift expected by Theorems 4 and 5 for all clients on the Internet. After this interval, the adversary cannot convince any clients of the validity of a revoked or expired certificate via network manipulation, so the CA may forget about its existence.

Appendix A

The Client/Server Protocol in ntpd

We present several ntpd vulnerabilities that stem from ambiguities in RFC5905. Figure A·1 is a (simplified) description[1] of the code used for the datagram protocol in ntpd v4.2.8p6 (the most recent release as of mid-April 2016).

Our attacks assume that client/server connections are unauthenticated, which is the default in ntpd and is the most common configuration in the wild (Section 5.5.1). Appendix A.1 presents our *Zero-0rigin Timestamp vulnerability* (CVE-2015-8138) that allows an off-path attacker to completely hijack an unauthenticated association between a client and its server, shifting time on the client. Appendix A.2 presents our *Interleaved Pivot vulnerability* (CVE-2016-1548), an extremely low-rate off-path denial-of-service attack. Importantly, both vulnerabilities affect ntpd clients operating in default mode, are performed from off-path, and require no special assumptions about the client's configuration. Both have been present in ntpd for seven years, since the first release of ntpd v4.2.6 in December 2009. In Section 5.4 we combine the interleaved pivot vulnerability with information-leaking NTP control queries and obtain a new off-path timeshifting attack.

A.1 Zero-0rigin timestamp vulnerability.

The zero-0rigin timestamp vulnerability allows an off-path attacker to completely hijack an unauthenticated client/server association and shift the client's time.

[1]Note that the actual ntpd code swaps the names of the `xmt` and `org` state variables; we have chosen the description that is consistent with the RFCs.

```
1  def receive( pkt ):
2      if pkt.T3 == 0:
3          flash |= test3  # fail test3
4      elif pkt.T3 == org
5          flash |= test1  # fail test1
6          return
7      elif broadcast == True:
8          ;           # skip further tests
9      elif interleave == False:
10         if pkt.T1 == 0:
11             xmt = 0
12         elif (xmt == 0 or pkt.T1 != xmt):
13             flash |= test2  # fail test2
14             if (rec !=0 and pkt.T1 == rec):
15                 interleave = True
16             return
17         else:
18             xmt = 0   # pass test2, clear xmt
19     elif (pkt.T1 == 0 or pkt.T2 == 0):
20         flash |= test3     # fail test3
21     elif (rec != 0 and rec != pkt.T1):
22         flash |= test2
23         return  # fail interleave test2
24
25     if interleave == False:
26         rec = pk.receive_time()
27     org = pkt.T3
28
29     if flash == True:
30         return
31     else
32         process( pkt )
33 return
```

Figure A·1: Simplified implementation of the datagram protocol from ntpd v4.2.8p6. The packet will not be processed if the `flash` variable is set. `interleave` variable is set when the host is in interleaved mode. Line 16 was introduced at ntpd v4.2.8p4 and 10-11 at ntpd v4.2.8p5.

Injecting 0rigin packets from off-path. In this attack, the attacker sends a spoofed mode 4 response packet to the target client. The spoofed packet has its origin timestamp T_1 set to zero, and its other timestamps T_2, T_3 set to bogus values designed to convince the target to shift its time, and its source IP set to that of the target's server. (The off-path attacker learns the server's IP via the reference ID, per footnote 6.) Now, consider how the target processes the received spoofed zero-0rigin timestamp packet:

(1) For ntpd v4.2.8p5 or v4.2.8p6, a spoofed zero-0rigin packet will always be accepted, because it passes through lines 10-11, which skip `TEST2` altogether. While the addition of lines 10-11 may seem strange, we suspect that they were added to handle the initialization of NTP's symmetric mode, which shares the same code path as client/server mode. Further discussion is in Appendix B.3.

(2) For ntpd v4.2.6 to v4.2.8p4, lines 10-11 of Figure A·1 were absent. Thus, the target will accept the spoofed packet when it does not have an outstanding query to the server. Why? When the target does not have an outstanding query to its server, its `xmt` variable is cleared to zero. Thus, when the spoofed zero-0rigin packet is subjected to `TEST2` (line 12), its origin timestamp (which is zero) will be compared to the `xmt` variable (which is also zero) and be accepted. The vulnerability arises because Figure A·1 fails to apply `TEST3`, which rejects packets with zero origin timestamp.[2]

Thus, an off-path attacker can send the target a quick burst of self-consistent zero-0rigin packets with a bogus time, and cause the target to shift its time. The spoofed zero-0rigin packets are always accepted in case (1), and usually accepted in case (2) because the target is unlikely to have outstanding query to its server. (In case (2), this follows because the server is queried so infrequently—NTP's polling intervals are at least 16 seconds long, but are often up to 15 minutes long.) After accepting the

[2] `TEST3` is applied in the fifth clause in Figure A·1, but a client will not enter this clause unless it is in interleaved mode.

burst of zero-0rigin packets, the target immediately shifts its time; in fact, the target shifts its time more quickly than it would under normal conditions, when legitimate responses arrive from the server at the (very slow) NTP polling rate (Section 5.2).

Experiment. On April 29, 2016, we performed a zero-0rigin timestamp attack on an ntpd v4.2.8p6 client. The target client uses the `-g` option on an operating system that restarts ntpd when it quits.[3] The target is configured to take time from a single server. The target starts and completes 15 timing exchanges with its server, averaging about one exchange per minute. We then attack, sending the target a spoofed zero-0rigin timestamp packet every second for ten seconds; these spoofed packets have T_3 as October 22, 1985 and T_2 as August 1, 2006. (This choice of T_2 sets $\delta \approx \psi \approx 0$, so we pass `TEST11`; see Section 5.2.) The target panics and restarts after the ninth spoofed packet. It then receives the tenth spoofed packet *before* it queries its server, and per the *reboot bug* in Appendix D, immediately shifts to 1985. (Our attack would still work even without this bug; the target would shift to 1985 after we sent it a few more packets.)

Eventually, the attacker decides to check if the attack has been successful by sending a mode 3 query to the target. By checking the transmit timestamp of the mode 4 response, the attacker realizes that the target is in 1985. Then, the attacker sets $T_2 = 0$ on his spoofed packets to maintain the target client in 1985. (This

[3]NTP's has a *panic threshold* that is 1000 seconds (16 mins). If the client gets a time shift that exceeds the panic threshold, the client quits. Thus, at first glance, it seems that that the worst an attacker can do is alter the client's clock by 16 minutes. However, as noted in (Malhotra et al., 2016), this panic behavior can be exploited. ntpd has a `-g` option that allows a client to ignore the *panic threshold* when it reboot; `-g` is the default ntpd configuration on many OSes including CoreOS Alpha (1032.1.0), Debian 8.2.0, Arch Linux 2016.05.01, etc. Moreover, many operating systems uses process supervisors (*e.g.,* systemd), which can be configured to automatically restart any daemons that quit. (This behavior is the default in CoreOS and Arch Linux. It is likely to become the default behavior in other systems as they migrate legacy init scripts to systemd.) Thus, an an attacker can circumvent the protections of panic threshold by sending the client a timeshift that exceeds the panic threshold, causing the NTP daemon to quit. The OS subsequently automatically reboots the NTP daemon. Now, if the NTP daemon is running with the `-g` option, it will ignore the panic threshold because it has just rebooted.

is necessary because the target client's T_4 is now October 22, 1985, so maintaining $\delta \approx 0$ so we pass `TEST11` requires $T_2 = 0$.) The attacker continues pelting the target with spoofed packets at a higher rate (≈ 1 packet/second) than that of the legitimate server response packets (≈ 1 packet/minute). The legitimate packets look like outliers (due, in part, to `TEST11`, Section 5.2) and the target sticks to the attacker's bogus time.

A.2 Interleaved pivot vulnerability.

We next consider a vulnerability introduced by NTP's interleaved mode, which is designed to allow for more accurate time synchronization.

What is interleaved mode? Recall that NTP uses timestamps on the packets to determine the offset θ between the client and the server. Because these timestamps must be written to the packet *before* the packet is sent out on the network, there is a delay between the time when the packet is 'formed' and the time when the packet is sent. This delay is supposed to introduce small errors in the offset. Interleaved mode eliminates this delay by spreading the computation of the offset (equation (6.2)) over *two* exchanges, rather than just one. In interleaved mode, hosts record timestamps for the moment that they actually transmit packet onto the network, and send them in the packet transmitted in the subsequent polling interval. Interleaved mode is not mentioned in RFC 5905 (Mills et al., 2010), but is implemented in ntpd. Mills (Mills, 2011) indicates that interleaved mode is intended for use on top of the broadcast or symmetric modes only.

We will exploit the following issues: (1) Interleaved mode shares the same code path as the client/server code (Figure A·1). (2) Interleaved mode *changes* the meanings of the values stored in the timestamp fields of an NTP packet. Importantly, the origin timestamp field of the mode 4 server response now contains T_4 from the

previous exchange (rather than T_1 from the current exchange). Thus, the usual `TEST2` no longer works; instead, there is 'interleaved `TEST2`' comparing the packet's origin timestamp to the `rec` variable, which stores T_4 from the previous exchange. (Line 21 in Figure A·1.) (3) A host *automatically switches* into interleaved mode when it detects that the host on the other side of the association is in interleaved mode. (Line 15 in Figure A·1.)

Interleaved mode as a low rate DoS vector. The implementation of interleaved mode in ntpd introduces a low rate denial-of-service attack. The vulnerability is introduced in line 14 of Figure A·1. Namely, if a server response packet fails the usual `TEST2`, the client subjects the packet to 'interleaved `TEST2`'. If the packet passes, the client sets the `interleave` variable and enters interleaved mode. Importantly, a client cannot escape from interleaved mode—there is no code path to clear the `interleave` variable.

Thus, an off-path attacker can inject a spoofed server response packet that passes 'interleaved `TEST2`' because its origin timestamp equals T_4 from the previous exchange. But how can the attacker learn T_4? It turns out that whenever an ntpd client updates its clock, it sets its reference time to be T_4 from the most recent exchange with the server to which it synchronized. This T_4 is sent out with every subsequent packet in the *reference timestamp* field.[4] Thus, to learn T_4, the off-path attacker first sends the client a regular mode 3 query, and learns the reference time from the client's response. If the target updated its clock in the previous exchange, the reference time will be T_4 from the previous exchange. The target will then react to the spoofed packet by switching into interleaved mode (Line 14 of Figure A·1). All subsequent legitimate server responses are rejected because they fail 'interleaved `TEST2`'.

Worse yet, this vulnerability also leads to a low-rate DoS attack that could be sprayed across the Internet (*e.g.,* using Zmap (Durumeric et al., 2013)). The attack

[4]Miroslav Lichvar noted that T_4 leaks in the reference timestamp.

leads to a DoS for *each one* of the target's servers. The attack works by repeating the process of (1) sending a timing query to the target to learn the IP of the server that the target synchronizes to, and its T_4 timestamp, and then (2) pivoting the target into interleaved mode for that server by sending a spoofed interleaved pivot packet. Thus, whenever the target synchronizes to a new server, the attacker will detect this (in step (1)) and DoS that new server as well (in step (2)). (This process is similar to the DoS by Spoofed Kiss-o'-Death attack from (Malhotra et al., 2016, Sec V.C).)

Timeshifting attacks. Section 5.4 shows how nptd's control query interface can be leveraged to turn the interleaved pivot vulnerability into a time-shifting attack (rather than just a DoS attack). Section 5.5.4 finds 700K vulnerable IPs.

Appendix B

Flaws in Symmetric Mode

Some of the vulnerabilities in NTP's client/server mode (mode 3/4) follow because it shares the same code path as NTP's symmetric mode (mode 1/2). Therefore, we now consider the security of NTP's symmetric mode. We identify several flaws in its specification in RFC5905 (including several off-path denial-of-service (DoS) attacks on unauthenticated symmetric mode, and several replay attacks (*i.e.,* on-path DoS attacks) on authenticated symmetric mode), explain how these flaws harm client/server mode, and conclude with recommendations.

B.1 Background: Symmetric mode.

In symmetric mode, two peers Alice and Bob can give (or take) time to (or from) each other via either ephemeral *symmetric passive* (mode 2) or persistent *symmetric active* (mode 1) packets. The symmetric active/passive association is preconfigured and initiated at the 'active' peer (Alice), but not preconfigured at the 'passive' peer (Bob). Upon arrival of a persistent mode 1 NTP packet from Alice, Bob mobilizes a new ephemeral association if he does not have one already. Because this is a potential security risk—an arbitrary attacker ask Bob to become its symmetric peer and start offering time to Bob—ntpd requires symmetric passive associations to be cryptographically authenticated by default. Active/active symmetric associations are also possible, where both peers are preconfigured with persistent associations. In this case, authentication is *not* required by default.

Symmetric mode has two additional quirks. First, each peer uses its own polling algorithm to decide when to respond to its peer. As such, Bob will not immediately respond to Alice upon receipt of her packet. (This is in contrast to the client/server mode, where servers immediately respond to queries.) Second, both peers perform TEST2 (and other tests) on the *same* volley of packets, and use the *same* packet timestamps to obtain timing samples.

B.2 Problems with bogus packets.

In both RFC5905 and ntpd, a host processes (mode 1 and 2) symmetric mode packets it receives using the same code used to process (mode 4) server response packets. Another look at this code in Appendix A of RFC 5905 (Figure 5·5) shows that the org state variable is updated even when a received packet *fails* TEST2. ntpd prior to v4.2.8p4 does this as well (Lines 16 and 27 in Figure A·1). But should a bogus packet really be allowed to update the client's state? We now explain why there is no easy answer to this question.

What if bogus packets do not update org? We first suppose that the org state variable is *not* updated upon receipt of a *bogus packet* (*i.e.,* a packet that fails TEST2). We show this leads to persistent failures in two cases:

1) Packet drop leads to persistent failure. In Figure B·1 Alice's second packet to Bob is dropped. After Alice's packet is dropped, Bob's $\texttt{org}_b$ state variable still stores the (now stale) time T_1. Bob uses T_1 as the origin timestamp of the packet he now sends to Alice. Alice drops this packet because its origin timestamp T_1 does not match her $\texttt{xmt}_a = T_5$ variable. Now this 'bogus' packet also does *not* update Alice's $\texttt{org}_a$ variable. Next, Alice sends a new packet to Bob at time T_9, using the (now stale) value $\texttt{org}_a = T_3$ as the new packet's origin timestamp. Now Bob drops the packet, because its origin timestamp T_3 does not match $\texttt{xmt}_b = T_7$. This continues

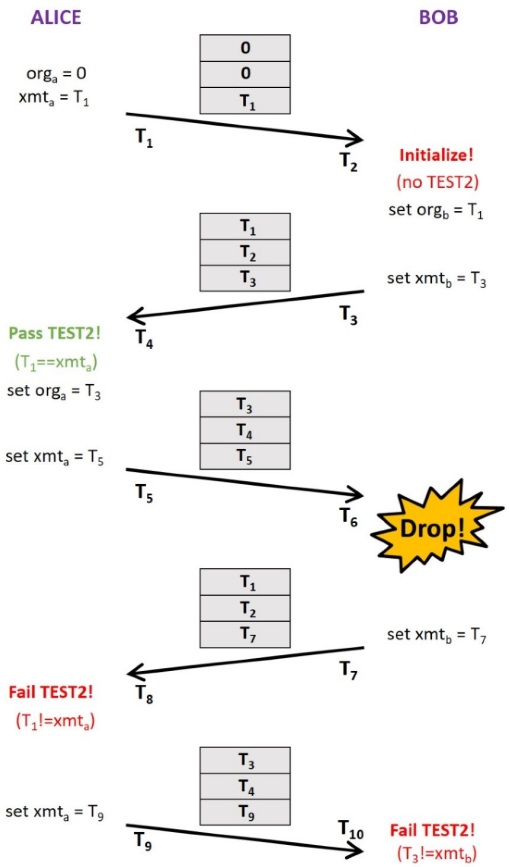

Figure B·1: Alice (left) exchanges symmetric mode packets with Bob (right). Each grey packet depicts the following fields (per Figure 5·4) in order: origin timestamp, receive timestamp, transmit timestamp. Alice's state variables $\texttt{org}_a$ and $\texttt{xmt}_a$ are shown on the left. Bob's state variables $\texttt{org}_b$ and $\texttt{xmt}_b$ are shown on the right. (Alice initializes the association by sending Bob an initialization packet, with origin and receive timestamps set to zero, and transmit timestamp set to Alice's sending time T_1. Alice writes T_1 to $\texttt{xmt}_a$. Bob receives the packet and copies the transmit timestamp T_1 from the packet to his $\texttt{org}_b$. When Bob's polling algorithm indicates he is ready to respond, he sends Alice a packet with origin timestamp T_1 copied from his $\texttt{org}_b$ and transmit timestamp T_3 equal to his time when he sent the packet. Bob then writes T_3 to his $\texttt{xmt}_b$. Upon receipt of Bob's packet, Alice performs TEST2, updates her state variables, computes offset, delay, *etc.*, and decides whether to update her clock. When her polling algorithm indicates that she is ready to respond, she constructs her next packet to Bob using her state variables in the same way Bob did.) In this Figure, Alice's second packet to Bob is dropped. If bogus packets (failing TEST2) do not update org, as shown here, then one dropped packet can cause persistent failure.

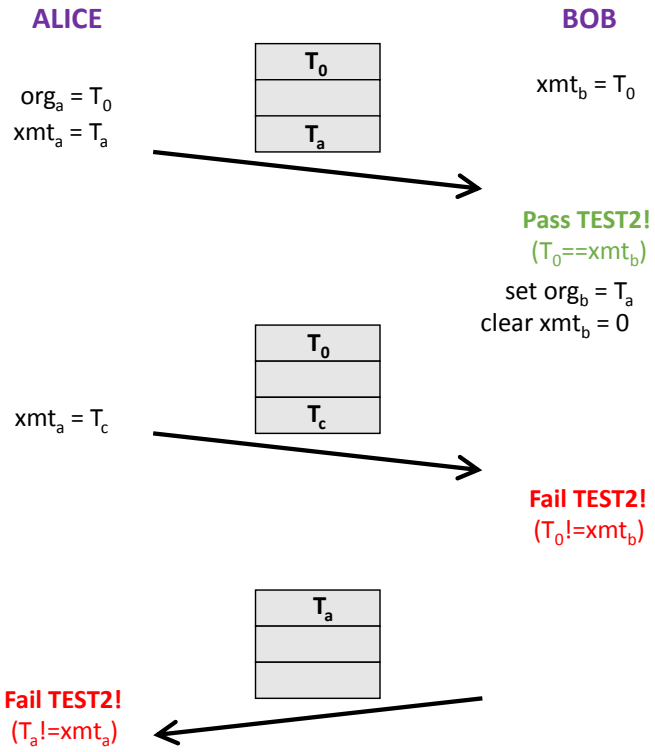

Figure B·2: Alice (left) exchanges symmetric mode packets with Bob (right). Alice sends two consecutive packets to Bob due to unsynchronized polling intervals. The first packet passes `TEST2`, but all subsequent packets fail `TEST2` on both peers, leading to persistent failure.

indefinitely, so all future packets fail `TEST2`.

2) Unsynchronized poll leads to persistent failure. We saw a similar failure happen naturally during an authenticated active/active symmetric association between peers Alice and Bob both running ntpd v4.2.8p6. This version of ntdp does *not* update `org` upon receipt of a bogus packet (because of the return added on line 16 in Figure A·1).

In Figure B·2 Alice was a symmetric peer with Bob. Bob was a symmetric peer with Alice, and also a client to an external server. Alice had a clock synchronization event that caused her to set her polling interval to 64 seconds. Meanwhile, Bob's polling interval was 128 seconds. Next, Alice sent Bob a packet with the correct origin timestamp T_0 expected by Bob. Bob accepted this packet and cleared $\texttt{xmt}_b$ and updates his $\texttt{org}_b = T_a$. However, Bob did not yet respond, since his polling

interval was longer than Alice's. In the meantime, Alice sent Bob another packet with this same origin timestamp T_0. (Alice sends the same T_0 because she has not yet received a new packet from Bob to cause her to update her $\texttt{org}_a$ variable.) This time Bob rejected the packet by TEST2 because he had cleared $\texttt{xmt}_b$. When Bob was ready to respond to Alice, he sent a packet with origin timestamp T_a matching to that in the first packet sent by Alice. (This is because Bob did not update his $\texttt{org}_b$ variable from the second rejected packet.) But Bob's packet failed TEST2 at Alice, because she was expecting origin timestamp T_c corresponding to the second packet. We are back in the persistent failure scenario of Figure B·1.

What if bogus packets do update org? The reader might now conclude that bogus packets *should* update org, as is required by Appendix A of RFC 5905. However, this leads to two denial-of-service attacks:

1) On-path denial-of-service for authenticated symmetric mode. Suppose that org can be updated by bogus packets that pass cryptographic validation (of the MAC) but fail TEST2. Consider an on-path attacker (who does not have the ability to drop/modify/delay packets) who attacks an *authenticated* symmetric association. (Note that symmetric active/passive associations are authenticated by default.) We show that this on-path attacker can parlay his ability to replay packets into the ability to (effectively) drop packets.

To do this, the on-path attacker replays any stale packet from Alice to Bob (1) after Alice sends Bob a legitimate packet but (2) before Bob sends his response as per Figure B·3. If these polling intervals are not synchronized, the attacker has plenty of time (*i.e.,* seconds or minutes) to perform this replay. The stale replayed packet overwrites Bob's $\texttt{org}_b$ variable to T_9. Thus, Bob responds to Alice with a packet whose origin timestamp is equal to the (stale) transmit timestamp T_9 from the stale replayed packet. This stale origin timestamp T_9 fails TEST2 at Alice. The attacker

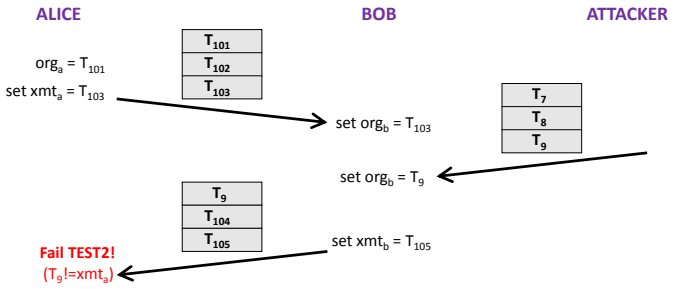

Figure B·3: Alice (left) exchanges symmetric mode packets with Bob (center). Attacker (right) is *on-path* for authenticated NTP and *off-path* for unauthenticated NTP. For the on-path authenticated DoS attack, the attacker's packet (timestamps T_7, T_8, T_9) is a replay of a stale packet sent from Alice to Bob. For the off-path unauthenticated DoS attack, the attacker's packet is spoofed.

can repeat this replay each time Alice sends Bob a packet, thus preventing Alice from ever synchronizing to Bob.

Note also this attacker need not be 'on-path' forever. Indeed, once the attacker gets his hands on a single stale packet sent from Alice to Bob, he can move off-path, and keep launching this attack forever by replaying this stale packet.

2) Off-path denial-of-service for unauthenticated symmetric mode. Suppose `org` can be updated by bogus packets that fail `TEST2`. We show how an off-path attacker can launch an identical attack on *unauthenticated* symmetric mode by spoofing (rather than replaying) a packet from Alice. This is a serious threat, since active/active symmetric associations are not cryptographically-authenticated by default.

We performed this attack on two ntpd v4.2.8p2 hosts Alice and Bob. (We use v4.2.8p2 because this implementation lets bogus packets update `org`.) Both Alice and Bob are preconfigured to be each other's symmetric active peer. Additionally, Bob is also preconfigured in client/server mode with four other servers. Upon restarting ntpd on both hosts, Bob gets synchronized to one of his servers in the very first exchange (per the reboot bug, see Appendix D). Alice sends symmetric active mode packets to Bob and gets back symmetric active response packets from Bob. After four exchanges with Bob, Alice synchronizes to Bob and indicates this by putting

Bob's IP address in the *reference ID* of her fifth packet. After two more exchanges, the off-path attack begins.

In symmetric mode, the time between a packet and its response is often up to several seconds (62 seconds in this experiment), giving our off-path attacker plenty of time to inject packets. So the attacker sends Bob a symmetric mode packet spoofed to look like it came from Alice; this query is sent after Alice sends her legitimate query to Bob, and before Bob sends his reply (per Figure B·3). Bob updates $\texttt{org}_b = T_9$ from attacker's bogus packet. Now Bob sends the response to Alice with origin timestamp T_9 corresponding to $\texttt{org}_b$. This packet fails TEST2 at Alice. The attacker continues to inject spoofed packets to Bob for the next 16 exchanges between Alice and Bob. Bob's responses fail TEST2 at Alice and so Alice never updates her clock.

Summary. Thus, we are between a rock and a hard place. Should bogus packets update org or not? Our recommendations are in Appendix B.4.

B.3 Problems with initialization.

Consider what happens if Alice reboots and sends Bob a packet initializing their association. Alice has no timing information, so this 'initialization packet' has $T_1 = T_2 = 0$ (as in the first packet in Figure B·1). If Bob did not reboot, he has xmt != 0. Now, if Bob performed TEST2 on the initialization packet, it would be dropped (because T_1 != xmt). Also, it would be dropped if Bob performed TEST3 (because $T_1 = T_2 = 0$). Thus, if the protocol is to tolerate a reboot, initialization packets cannot be subject to TEST2 or TEST3.

Denial-of-service via initialization packets. We use the fact that TEST2 cannot be performed on initialization packets to perform DoS attacks identical to those in Appendix B.2. We can perform on-path DoS attacks on authenticated symmetric associations by replaying initialization packets (instead of replaying stale packets).

Off-path DoS attacks on unauthenticated symmetric mode can also be accomplished by spoofing initialization packets (rather than spoofing arbitrary packets); notice that spoofing initialization packets is trivial because they do not contain any unpredictable information. Importantly, both of these DoS attacks exist *regardless of* whether bogus packets update `org` or not (Appendix B.2).

Impact on client/server mode. The initialization of symmetric mode requires that `TEST2` and `TEST3` are not performed on a received packet with a zero-origin timestamp. However, this is at odds with the security of client/server mode. Unfortunately, however, client/server and symmetric modes share the same code path. ntpd deals with symmetric mode initialization using Lines 10-11 in Figure A·1, which clears `xmt` and skips `TEST2` if a received packet has a zero-origin timestamp. These lines of code, however, create the zero-0rigin timestamp vulnerability in client/server mode (Appendix A.1). Meanwhile, RFC 5905 Appendix A deals with this by not performing `TEST3` (Figure 5·5). However, because `TEST3` is not performed, the `xmt` variable cannot be cleared, creating the query-replay vulnerability (Section 5.3.2).

B.4 Symmetric Mode: Choose your poison!

Many problems in symmetric mode occur because both peers update their state variables (`org`, `xmt`) and collect timing samples $(\theta_i, \delta_i, \psi_i)$ from the *same* volley of packets. Per the discussion in Appendix B.2, we cannot see how to fix this while maintaining a single volley of packets between peers. One drastic suggestion is to require *two* distinct volleys, where each peer is a server in one volley, and is a client in the other (using one of the protocols described in Section 5.6.1). However, this is not backwards compatible, as both peers involved in association must simultaneously make this change. Thus, an (unsatisfying) band-aid solution could involve:

1. To prevent the persistent failure problem of Appendix B.2, allow packets failing

`TEST2` to update `org`. However, this enables off-path DoS attacks.

2. To prevent off-path DoS attacks, we suggest mandatory cryptographic authentication in symmetric mode (for both active/active and active/passive).

3. Even so, symmetric peers that use cryptographic authentication are still vulnerable to DoS attacks, so we also suggest monitoring to detect excessive number of bogus packets (Appendix B.2).

4. Monitoring should also be used to detect excessive number of initialization packets, since these also lead to DoS (Appendix B.3).

5. Finally, symmetric peers should ensure that they run `TEST2` against an origin timestamp that contains 32 bits of randomness. This can be done with a receive function as in Figure 5·6 and a sending function in Figure 5·7.[1]

[1] Because both peers update their state variables and collect timing samples from the same volley of packets, symmetric mode must preserve the semantics of the origin timestamp. Thus, in symmetric mode we cannot replace the origin timestamp with a random 64-bit nonce per Figure 5·8.

Appendix C

On-path Query Replay Attack

Replays of the client's query are a problem because they harm the accuracy of time synchronization. We demonstrate this with an on-path query replay attack on a target host in our lab. We were able to degrade the accuracy of the target's time synchronization from 4×10^{-5} seconds (on average) to 2.7 seconds (on average). We show the client's offset (*i.e.,* the distance between the client's clock and the server's clock per equation (6.2)) under normal and attack conditions in Figure C·1; the attacked client's accuracy is 5 orders of magnitude worse.

Experiment. We modified the source code for ntpd v4.2.8p2 to make it vulnerable to client query replays. Specifically, we deleted the line of code that cleared `xmt` when a response passed `TEST2`. We then preconfigured our modified ntpd client with one server. Every time the client sent a query to its server, our on-path attacker captured the query, and replayed it to the server once per second, until the client sent a new query. We repeated this for every query sent by the client. The resulting offset in Figure C·1 was computed per equation (6.2) with T_1, T_2, T_3 taken from the NTP packet timestamps on server responses sent in response to real client queries (not replayed queries) and T_4 taken from the response packet's arrival time at the client. We repeat this experiment on the same client and server machine but without a replay attack.

Why does accuracy degrade? This follows because the replayed queries cause the server to respond with a stale origin timestamp T_1. Suppose that t seconds elapse

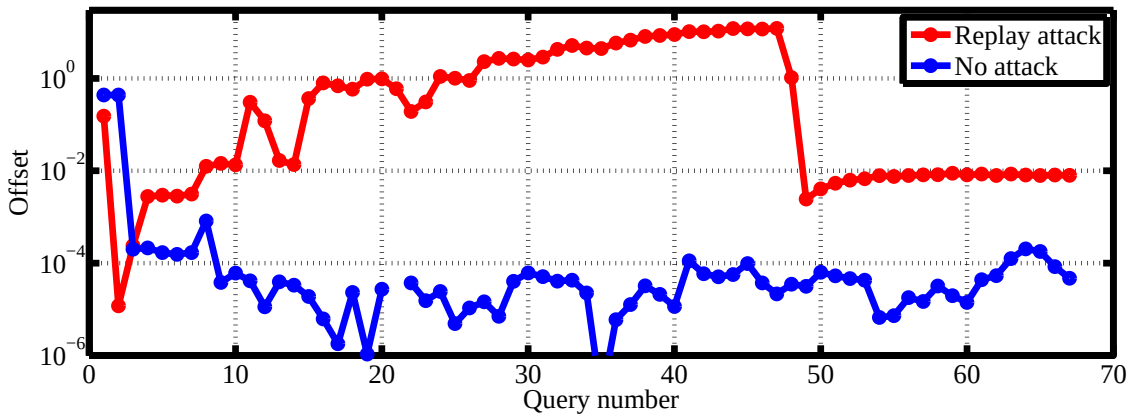

Figure C·1: Query replay attack on modified version of ntpd v4.2.8p2. Time synchronization on the attacked client degrades by 10^5.

since the client's most recent query. If the attacker now replays the client's query, the packet timestamps in the server's response will be such that $T_2 - T_1 \approx t$ seconds and $T_4 \approx T_3$ seconds, resulting in a timing sample with offset $\theta \approx \frac{t}{2}$ seconds per equation (6.2). As t grows during the polling interval, the offset in the timing sample grows as well. Thus, when the client uses these sampled offsets to set its clock, it miscalculates the discrepancy between its local clock and that of the server, resulting in the inaccuracies in Figure C·1. Thus, this query replay attack has similar effect to a delay attack (Mizrahi, 2012a).

Appendix D

Reboot Bug

Our experiments show that, upon reboot, an ntpd v4.2.8p6 client updates its local clock from the very first response packet it receives from *any* of its preconfigured servers. This is CVE-2016-7433. We now explain why this is a security vulnerability.

What does the RFC say? When describing the algorithm used for clock updates, the pseudocode in Appendix A.5.2 of RFC5905 has a comment that states "select the best from the latest eight delay/offset samples". Also, a client configured with multiple servers is supposed to choose the 'best' server from which it will take time. Section 5 says: "The selection algorithm uses Byzantine fault detection principles to discard the presumably incorrect candidates called "falsetickers" from the incident population, leaving only good candidates called "truechimers"." We argue that this bug disables Byzantine fault tolerance upon reboot.

Experiment. We set up an ntpd v4.2.8p6 client preconfigured with five pool servers. Upon reboot, the client sends server Alice a mode 3 query with *reference id* 'INITALIZATION' (indicating that it is unsynchronized) and *reference time* 'NONE' (expected behavior upon reboot). Another such query is made to Bob. Bob's response arrives first, followed by the response from Alice. Next the client sends a query to server Carol. The *reference id* field in this new query is Bob's IP, and the *reference time* is set to a time *before* the response was received from Alice. We therefore see that the client updates his clock upon receipt of his first response packet (from Bob), without considering the contributions of servers Alice, Carol, Dave and Frank.

Implications. Thus, on reboot, Byzantine fault tolerance is disabled, and the client is at risk of taking time from bad timekeepers. This issue becomes even more serious when the panic threshold is disabled upon reboot when a client is configured with `-g` option. (This is the default on many OSes, see footnote 3.) Thus, if a bad timekeeper's response arrives first, a `-g` client will immediately accept huge, potentially bogus, update to its clock.

Worse yet, an off-path attacker can exploit this, along with other bugs, in order to perform a low-rate time-shifting attack. The attacker first learns the IP of one of the target's preconfigured servers, using the trick per footnote 6. Then, the off-path attacker sends some 'packet-of-death' that crashes ntpd.[1] If the OS reboots ntpd, then the target restarts with the panic threshold disabled. The attacker now injects a single spoofed server response with (1) zero origin timestamp (per Appendix A.1), (2) the legitimate server as the source IP, and (3) some huge (incorrect) time offset. The client accepts the response even before it queries its legitimate servers, and adjusts its clock to the attacker's bogus time. The low rate of this attack—it requires only three packets—also means it could be sprayed across the Internet.

Recommendation. An NTP client should be compliant to the RFC specifications even upon reboot, and adjust its clock only after multiple successful exchanges with each of its timeservers.

Where is the bug introduced? The bug is introduced in ntpd v4.2.7p385 (released August 18, 2013) and exists in all the following versions, upto and including ntpd v4.2.8p7 (released April 26, 2016). The definition for *Root distance* (λ) in the variable *dtemp* in file `ntp_proto.c` was changed between ntpd v4.2.7p384 (Figure D·2) and ntpd v4.2.7p385 (Figure D·1). This change (Lines 2966-2967 in Figure D·1) introduced the bug. However, this change violates compliance with the definition of *Root distance* in RFC5905 which defines it as in Figure D·2.

[1] For instance, CVE-2016-7434 or CVE-2016-9311.

```
2965   dtemp = (peer->delay + peer->rootdelay) / 2
2966           + LOGTOD(peer->precision)
2967           + LOGTOD(sys_precision)
2968           + clock_phi * (current_time - peer->update)
2969           + peer->rootdisp
2970           + peer->jitter;
```

Figure D·1: Lines 2965-2970 in ntpd v4.2.7p385

```
2933   dtemp = (peer->delay + peer->rootdelay) / 2 + peer->disp
2934            + peer->rootdisp + clock_phi * (current_time - peer->update)
2935            + peer->jitter;
```

Figure D·2: Lines 2933-2935 in ntpd v4.2.7p384

Tested Versions.ntpd v4.2.7p384, ntpd v4.2.7p385, ntpd v4.2.8p6, ntpd v4.2.8p7, ntpd v4.2.8p8. This part of code remains the same in all the versions beginning ntpd v4.2.87p385.

Patch: Replacing code in Figure D·1 with that in Figure D·2 mitigates the bug. We successfully patched ntpd v4.2.8p7 (lines 3447-3453). To confirm, we ran the experiment with the patched version of ntpd v4.2.8p7 with the same setup as above. The test client updates its local clock after obtaining four timing samples from its servers.

Appendix E

Disclosure and Subsequent Developments

This research was done against ntpd v4.2.8p6, which was the latest version of ntpd until April 25, 2016. Since that date, three new versions of ntpd have been released: ntpd v4.2.8p7 (April 26, 2016), ntpd v4.2.8p8 (June 2, 2016), and ntpd v4.1.8p9 (November 22, 2016). We summarize our disclosure timeline and the impact of our research results on new releases of ntpd as follows:

Report. This report was first disclosed on June 7, 2016 and Section 4.4 was revised on July 29, 2016. The report was last edited, for clarity and style, on August 26, 2019.

Zero-0rigin timestamp vulnerability (CVE-2015-8138, CVE-2016-7431, Appendix A.1). This vulnerability was disclosed in October 2015 (prior to ntpd v4.2.8p4) but unfortunately still existed in v4.2.8p8. (This is because Lines 10-11 of Figure A·1 were still present in ntpd v4.2.8p8, likely in order to process initialization packets in symmetric mode, see Appendix B.3.) The vulnerability has been fixed in ntpd v4.2.8p9.

Interleaved pivot vulnerability (CVE-2016-1548, Appendix A.2). Following disclosure of this vulnerability in November 2015[1], ntpd v4.2.8p7 was patched so that clients do not automatically switch into interleaved mode by default. Now, clients do this only if the option '`xleave`' is set with a `peer`, `server` or `broadcast` configuration command.

[1] Also the concurrent disclosure by Miroslav Lichvar.

Leaky control queries (CVE-2015-8139, Section 5.4). This vulnerability was first disclosed in October 2015, but ntpd v4.2.8p9 still accepts control queries from arbitrary IPs by default. Users must configure the `noquery` option to change this default. The leaky control queries we describe are also mentioned in a new Internet draft (Mills and Haberman, 2016). We have worked with the authors of (Mills and Haberman, 2016) to add a security considerations to this document.

Origin timestamp randomization (Section 5.6). Cryptographic randomization of the origin timestamp has not yet been incorporated into ntpd.

Bogus packets in symmetric mode (Appendix B.2). ntpd v4.2.8p7, ntpd v4.2.8p8 and v4.2.8p9 allow bogus packets (that fail `TEST2`) to update the `org` state variables.[2] Thus, the 'transient failure to persistent failure' from Appendix B.2 is no longer present, but the two denial of service vulnerabilities in Appendix B.2 are present. This issue was disclosed on June 7, 2016.

DoS via initialization packets in symmetric mode (Appendix B.3). This was first disclosed on June 7, 2016. These flaws were still present in v4.2.8p9.

Reboot bug (CVE-2016-7433, Appendix D). We first disclosed this issue in August 2015, and provided a full analysis on June 7, 2016. This bug was still present in v4.2.8p8 and fixed in v4.2.8p9.

[2]This change was probably done in response to NTP Bug2952 reported by Michael Tatarinov and made public on April 26, 2016 (concurrently with our work). The bug report states only that "symmetric active/passive mode is broken" (Tatarinov,).

Appendix F

Security Analysis

In this appendix, we provide our formal security model and prove that the protocols in Appendix 5.6.1 are secure against off-path attackers when NTP is unauthenticated and against on-path attackers when NTP is authenticated.

F.1 Model

Our model focuses on the description of an honest party called the *network* $\mathcal{N}$ that delivers packets and orchestrates the execution of several NTP exchanges akin to the environment in a UC protocol (Canetti, 2001).

Parties. We suppose there are ℓ honest parties $\mathcal{P}_1, \ldots, \mathcal{P}_\ell$, where $\mathcal{P}_i$ denotes the IP address of the i^{th} party, who collectively perform many pairwise client/server exchanges. A single party $\mathcal{P}_i$ may act in both the client and server roles in different exchanges. There is also single attacker $\mathcal{A} = \mathcal{P}_0$ who may also have honest client/server exchanges, but has other goals and powers as well. Parties send packets through a *network* $\mathcal{N}$.

Packets. We model a packet as a tuple of the form (h, m, t), where $h = (\mathsf{IP\_src}, \mathsf{IP\_dst})$ contains the source and destination IPs, t is a MAC tag optionally appended to the packet, and m contains the remaining fields of the packet that are authenticated by the MAC tag.

Packet delivery. The *network* $\mathcal{N}$ maintains a counter step to orchestrate the flow of communication. Informally, one may think of step as the wall-clock time from $\mathcal{N}$'s

point of view. During each step of the step counter, each honest party may receive, process, or transmit one packet.

We require that honest parties have sufficient time to process every packet received; put another way, attacks that flood an honest party with packets in order to deny service are out of scope. Formally, we model this property by (1) restricting $\mathcal{N}$ to deliver at most one packet per step of the counter to each party, and (2) incrementing the counter in integer multiples of L/R, where R denotes an upper bound on the bandwidth (bps) of honest parties when ingesting NTP packets of length L bits. Here, $L = 720$ bits for unauthenticated NTP packets and $L = 720 + 2n$ bits for NTP packets authenticated with a MAC of length $2n$.

The network imposes a constant latency δ on packet delivery. Specifically, if $\mathcal{N}$ receives a packet from an honest party when the counter is step, $\mathcal{N}$ holds the packet in a queue and assigns a value deliver $=$ step $+ \delta$ to the packet. When step $==$ deliver, the packet is transmitted. We stress that the attacker $\mathcal{A}$ does *not* have the power to modify, delay, or drop packets between honest parties.

Race conditions. Unlike the honest parties, attacker $\mathcal{A}$ may specify the deliver value for all packets she sends. However, as per constraint (1) above, this value must be distinct from the deliver values of all other packets in $\mathcal{N}$'s queue destined for IP_dst.

This power allows $\mathcal{A}$ to win all race conditions.[1] As such, our model allows attakcer $\mathcal{A}$ to send an honest party up to $2\delta\frac{R}{L}$ packets in the duration of an NTP exchange, since $\mathcal{A}$ can request the delivery of one packet for each step of the counter, while the two messages in an NTP exchange are each subject to a longer delay δ.

Additionally, this power also encapsulates the real-world uncertainty in packet delivery. So far, our model assumes that all packets encounter a constant delay. In reality, we often have at most a rough upper-bound on network latency, and we want

[1] We remark that this capability is unrealistically powerful for an off-path attacker, who cannot observe honest packet transmissions.

for NTP's security guarantee to hold for any distribution of packet latency times that fit within this bound. Rather than formalizing a network latency distribution within our model, we opt for the simpler approach of letting attacker $\mathcal{A}$ "speed up" packet arrivals using its power to determine $\mathsf{deliver}$ for its packets.

Transcripts. A *transcript* is a list of NTP client/server exchanges, formally written as a set of tuples

$$(\mathsf{start}, i, t_c, j, t_s)$$

each indicating that an exchange between client $\mathcal{P}_i$ with local time t_c and server $\mathcal{P}_j$ with local time t_s starts at step start of the step counter. We stress that $\mathcal{A} = \mathcal{P}_0$ may legitimately engage in NTP exchanges in the transcript specification.

Network $\mathcal{N}$ enforces execution of the transcript by informing parties that they should begin a client/server exchange. When $\mathcal{N}$'s counter is $\mathsf{step} == \mathsf{start}$ for tuple $(\mathsf{start}, i, t_c, j, t_s)$ in the transcript ts, $\mathcal{N}$ sends a 'go message' (client, i, t_c) to $\mathcal{P}_i$, after which an honest party $\mathcal{P}_i$ immediately sets her local clock to t_c, runs the protocol in Figure 5·7 or Figure 5·8 resulting in a mode 3 query packet $((\mathcal{P}_i, \mathcal{P}_j), m, t)$ to $\mathcal{P}_j$ through $\mathcal{N}$. After a delay of δ, $\mathcal{N}$ sends a 'go message' (server, i, t_s) to the $\mathcal{P}_j$ and also delivers the $\mathcal{P}_i$'s query packet, and honest $\mathcal{P}_j$ responds assuming that her local clock is set to time t_s.

The transcript must be consistent with our "no flooding" rule that limits each party to receiving 1 packet (plus perhaps a 'go message') per step. As a consequence, two exchanges involving the same client cannot have the same start counter. Additionally, a party cannot simultaneously be a server at counter start and a client at counter $\mathsf{start} + \delta$.

Interacting with the network. The network $\mathcal{N}$ starts by receiving ts and then choosing and dispersing secret keys for each pair of parties $\{\mathsf{sk}_{i,j} : i, j \in \{0, 1, \ldots, \ell\}\}$. The honest parties $\mathcal{P}_i$ receive these keys and initialize their $\mathtt{xmt}_j$ state variables for every

other party $j \neq i$. The game then begins with $\mathcal{N}$ in control and the counter step initialized to 0.

If $\mathcal{N}$'s counter step equals either (1) the start value of a tuple in the transcript ts or (2) the $\mathsf{deliver}$ value of a packet in its queue, then $\mathcal{N}$ delivers the appropriate packet or 'go message' and cedes control to the honest party. The honest party starts computing when it receives the packet or 'go message'.

Once an honest party finishes its computation and possibly transmits a new packet $((\mathsf{IP\_src}, \mathsf{IP\_dst}), m, t)$ to $\mathcal{N}$, the honest party cedes control back to $\mathcal{N}$. Next, the network $\mathcal{N}$ instantly reveals $[(\mathsf{IP\_src}, \mathsf{IP\_dst}), \mathcal{L}(m, t)]$ to the attacker, where $\mathcal{L}$ is a *leakage function*. $\mathcal{N}$ then cedes control to $\mathcal{A}$, who may perform arbitrary computations and optionally transmit a packet of its own. When $\mathcal{N}$ regains control, it increments step and repeats the process. This model implicitly forbids $\mathcal{A}$ from dropping, modifying, or further delaying packets; instead, every packet is delivered intact to $\mathsf{IP\_dst}$ after delay δ.

Leakage. The *leakage function* $\mathcal{L}$ models the information available to an on-path or off-path attacker. Specifically, $\mathcal{L}$ equals the identity function for an on-path attacker (*i.e.,* m and t are revealed perfectly) and the zero function for an off-path attacker (*i.e.,* m and t are perfectly hidden).

Spoofing. Network $\mathcal{N}$ never validates $\mathsf{IP\_src}$ in a transmitted packet. This allows attacker $\mathcal{A}$ to send packets with a spoofed source IP $\mathsf{IP\_src}$. Meanwhile, honest parties always use their true $\mathsf{IP\_src}$. Additionally, if $\mathcal{A}$ spoofs a query packet on behalf of a client $\mathcal{P}_i$, we observe that $\mathcal{N}$ lacks a timestamp t_s^* to deliver to the honest server $\mathcal{P}_j$ along with the query packet. We choose t_s^* as follows: if $\mathcal{A}$'s spoofed packet occurs during an honest NTP transaction between parties i and j, then $\mathcal{N}$ sends the same timestamp that the honest transaction uses; otherwise, $\mathcal{A}$ may choose t_s^* arbitrarily and inform $\mathcal{N}$ of its choice.

F.1.1 Soundness guarantee.

Without an attacker, the results of honest parties' NTP exchanges are completely defined by the transcript. Formally, clients update their local *state* which includes the set of servers they are willing to query, the state variables (*i.e.,* $\texttt{xmt}_j$ and $\texttt{org}_j$) used in exchanges with each server $\mathcal{P}_j$, and the resulting set of validated timing samples (including delay δ, offset θ, jitter ψ, dispersion ϵ per the equations in Section 5.2). $k\text{-}\mathsf{state}_i(\mathsf{ts}, \mathsf{step})$ is the state of party $\mathcal{P}_i$ during an execution of the transcript ts while $\mathcal{N}$'s counter is step, and contains the results of the k most recent exchanges with each server.

$\mathcal{A}$'s objective is to tamper with k consecutive timing samples that some honest client $\mathcal{P}_i$ stores in its state corresponding to interactions with a single server $\mathcal{P}_j$.[2] Hence, we let $k\text{-}\mathsf{state}_i^{\mathcal{A}}(\mathsf{ts}, \mathsf{step})$ denote the state of party $\mathcal{P}_i$ during a game where the attacker $\mathcal{A}$ is present. Of course, if $\mathcal{P}_i$ voluntarily chooses to query $\mathcal{A}$ as its server, then $\mathcal{A}$ can significantly influence $\mathcal{P}_i$'s state. The soundness guarantee effectively states that $\mathcal{A}$ can do no more than this.

However, there is one type of modification that we cannot hope to rule out. Consider the effect of $\mathcal{A}$ "preplaying" honest packets: that is, submitting a packet that is identical to one in $\mathcal{N}$'s queue but with an earlier arrival time. This action is very likely to affect the state of honest parties, albeit in a bounded manner. It may reduce delay measurements δ from their upper bound of 2δ, but never increase them. Similarly, each offset θ may increase or decrease by at most δ. Finally, jitter ψ may be altered slightly, likely by far less than the bound accepted by `TEST11`. Due to their limited, unavoidable effects, we consciously opt to ignore preplay attacks in the following soundness definition to simplify our discussion.

[2]Note that the definition 'NTP exchanges should not fail' does not hold because exchanges may fail even without an attacker. As one example, consider a client who initiates two exchanges with the same server in rapid succession, *i.e.,* the client's second query is sent before she receives a response to the first query. Then `TEST2` will fail for the server's first response.

Definition F.1.1 (Soundness)**.** NTP is (k, ϵ)-*sound* on transcript ts if for all *resource-bounded* attackers $\mathcal{A}$ who never preplay packets from honest parties, and for all parties $\mathcal{P}_i$ who do not query $\mathcal{A}$ as an NTP server,

$$\Pr[\exists\ \mathsf{step} \text{ s.t. } k\text{-}\mathsf{state}_i^{\mathcal{A}}(\mathsf{ts}, \mathsf{step}) \neq k\text{-}\mathsf{state}_i(\mathsf{ts}, \mathsf{step})] < \epsilon.$$

This inequality must hold for *all* k components of the state. The probability is taken over the randomness of all parties and $\mathcal{N}$'s choice of shared secret keys.

F.2 Soundness against off-path attackers.

We now prove Theorem 1 of Appendix 5.6.3, which states that the protocols in Appendix 5.6.1 are sound against an off-path attacker $\mathit{off}\mathcal{A}$. Theorem 1 follows largely from the entropy E present in the origin timestamp. We do not require NTP packets to be authenticated.

The theorem holds as long as randomness is produced from a cryptographically-strong random number generator (RNG), and that, upon reboot, honest parties initialize their $\mathtt{xmt}_j$ variables for each server to a 64-bit number generated by their RNG.

Let $\mathit{off}\mathcal{A}$ be any off-path attacker, and let ts be any transcript that involves ℓ honest parties, a maximum of τ exchanges involving any single client-server pair, and a maximum of s trusted servers per client.

Let i^* be any client who does not query $\mathit{off}\mathcal{A}$ as server. We say that the protocol described in Figure 5·6, 5·7 randomizes the sub-second granularity of the expected origin timestamp, while the protocol in Figure 5·6, 5·8 randomizes the entire expected origin timestamp.

We use a sequence of games to prove that $\mathit{off}\mathcal{A}$ tampers the state of $\mathcal{P}_{i^*}$ with probability at most $\epsilon_{\mathit{off}\mathcal{A}}$.

Game G_0. This is the real interaction of $\mathit{off}\mathcal{A}$ with the honest parties $\mathcal{P}_1, \ldots, \mathcal{P}_l$ and

the network $\mathcal{N}$. For ease of notation, we denote the probability that *off*$\mathcal{A}$ breaks the soundness of game G_0 by $\Pr^0_{off\mathcal{A}}$.

Game G_1. This game is identical to G_0, except that $\mathcal{P}_{i^*}$'s pseudorandom number generator is replaced with a truly random number generator. By definition, the probability that anybody (in particular *off*$\mathcal{A}$) notices this change is at most $\mathcal{A}$(RNG). Hence, $\Pr^0_{off\mathcal{A}} - \Pr^1_{off\mathcal{A}} \leq \mathcal{A}(\text{RNG})$.

Game G_2. This game is identical to G_1, except that we abort the execution if *off*$\mathcal{A}$ sends a spoofed packet (*i.e.,* one for which *off*$\mathcal{A}$ claims an $\mathsf{IP\_src}$ different than her own) involving client $\mathcal{P}_{i^*}$ and some server $\mathcal{P}_j$ such that the spoofed packet's origin timestamp matches $\mathcal{P}_{i^*}$'s state variable $\texttt{xmt}_j$. Importantly, *off*$\mathcal{A}$ has no chance of winning game G_2 (that is, $\Pr^2_{off\mathcal{A}} = 0$) because its spoofed packets always fail TEST2. In order to demonstrate that any client $\mathcal{P}_{i^*}$ distrusting *off*$\mathcal{A}$ properly refuses all of the attacker's spoofed packets while also accepting all of the honest servers' packets (*i.e.,* computes the desired value k-$\mathsf{state}_i(\mathsf{ts}, \mathsf{step})$ at all steps), it only remains to prove that the probabilities of winning G_1 and G_2 are close.

There are two conditions that cause Game G_2's abort condition to trigger:

- Client $\mathcal{P}_{i^*}$ is engaged in an NTP exchange with server $\mathcal{P}_j$ at the moment the spoofed packet is received, and the spoofed packet's origin timestamp matches that of the transmit timestamp in honest client's query.

- Client $\mathcal{P}_{i^*}$ is *not* engaged in an NTP exchange with server $\mathcal{P}_j$ at the moment the spoofed packet is received, and the spoofed packet's origin timestamp matches the client's randomly-chosen $\texttt{xmt}_j$ value.[3]

In the first case, we know that the client's choice of the origin timestamp expected in the mode 4 response packet (*i.e.,* pkt.T3 in Figure 5·7) ensures that $\texttt{xmt}_j$ has

[3]Recall that $\mathcal{A}$ may choose the server's response time t^*_s arbitrarily in this case, which would have immense power if $\mathcal{A}$ could get the spoofed client to accept the response packet.

$E = 32$ bits of entropy if the sub-second granularity of the timestamp comes from $\mathcal{P}_i$'s RNG or $E = 64$ bits of entropy if the entire origin timestamp is randomly chosen (*i.e.,* `pkt.T3` in Figure 5·8). When targeting a particular server $\mathcal{P}_j$, *off*$\mathcal{A}$ can send $\frac{R\delta}{360}$ packets to each honest party during an NTP exchange between the target client $\mathcal{P}_{i^*}$ and the server $\mathcal{P}_j$, each of which influences $\mathcal{P}_{i^*}$'s state with probability at most 2^{-E} where E is the number of bits of entropy in the origin timestamp. Hence, *off*$\mathcal{A}$'s ability to impact $\mathcal{P}_{i^*}$'s state during the NTP exchange is at most $Q_d = 2^{-E}\frac{R\delta}{360}$.

In the second case, *off*$\mathcal{A}$ can send $T = 2^p R/720$ packets to each party in the interval between two successive exchanges (where p corresponds to the polling interval). Each packet succeeds in altering $\mathcal{P}_{i^*}$'s state with probability $Q_b = 2^{-64}$ because $\texttt{xmt}_j$ has 64 bits of entropy.

Finally, Lemma 1 below states that *off*$\mathcal{A}$ can influence the state of k consecutive exchanges between client $\mathcal{P}_{i^*}$ and server $\mathcal{P}_j$ with probability at most $(k+1)\cdot(kQ)^k$, where $Q = \max\{Q_d, TQ_b\}$. Additionally, there are τ possible locations for this run of k successes to start, and s possible servers whose state may be attacked. In total, we find that:

$$\Pr^1_{\mathit{off}\mathcal{A}} - \Pr^2_{\mathit{off}\mathcal{A}} \leq (k+1)s\tau \cdot (kQ)^k$$

In practice, we claim that $Q_d > TQ_b$ if entropy $E = 32$:

$$2^{-32} \cdot \frac{R\delta}{360} > 2^{-64} \cdot 2^p \frac{R}{720}$$
$$2^{33}\delta > 2^p$$

With the maximum poll value $p = 17$ permitted by NTP (Mills et al., 2010), this reduces to the claim that $\delta > 2^{-16} \approx 10^{-5}$ seconds, which is the time required for light to travel about 3 miles. So, our inequality is reasonable unless the client and server are physically co-located but still using a large polling value. Conversely, we claim $TQ_b > Q_d$ if entropy $E = 64$: this claim reduces to the statement that $2^p > 2\delta$, which

holds since RFC5905 constrains poll $p \geq 4$ while network delays δ do not exceed 16 seconds in practice.

All that remains is to prove the following combinatorial statement relating the probabilities of success during and between exchanges.

Lemma 1. *Let Q_d denote the probability that an attacker $\mathcal{A}$ successfully impacts the state of client $\mathcal{P}_{i^*}$ during an NTP exchange, Q_b denote the probability that each packet by $\mathcal{A}$ in between NTP exchanges impacts $\mathcal{P}_{i^*}$'s state, and let $T = 2^p \cdot \frac{R}{720}$ denote the number of packets that $\mathcal{A}$ may send to each party in between NTP exchanges. Then, the probability that $\mathcal{A}$ impacts k state observations in a row, beginning with a specified exchange, is at most $(k+1) \cdot (kQ)^k$, where $Q = \max\{Q_d, TQ_b\}$.*

$\mathcal{A}$ may compromise a total of k states either during or between exchanges. Let $c \in \{0, 1, \ldots, k\}$ denote the number of consecutive NTP exchanges (with a specified starting point) that $\mathcal{A}$ plans to compromise; clearly, she may do so with probability Q_d^c. Additionally, $\mathcal{A}$ must also inject a total of $k - c$ state measurements over the course of $c + 1$ intervals between these NTP exchanges. Here, each packet is an independent Bernoulli random variable that successfully impact's the client's state with success probability Q_b. The total number of between-exchange successes (i.e., the sum of the $(c+1)T$ Bernoullis) is distributed as a binomial random variable, hence the probability of $k - c$ total successes is at most $\binom{(c+1)T}{k-c} \cdot Q_b^{k-c}$.

In total, $\mathcal{A}$ succeeds at compromising c NTP exchanges and successfully injecting state $k - c$ times in between these exchanges with probability at most

$$\binom{(c+1)T}{k-c} Q_d^c Q_b^{k-c} \leq (kQ)^k,$$

where the inequality follows from the bound $\binom{x}{y} \leq x^y$. The lemma then follows by summing the probabilities of success for the $k + 1$ choices of c.

F.3 Soundness against on-path attackers.

We now prove Theorem 2 of Appendix 5.6.4, which states the protocols in Section 5.6.1 are sound against an on-path attacker $on\mathcal{A}$ as long as NTP packets are authenticated, randomness is produced from a cryptographically-strong RNG, and honest parties initialize their $\texttt{xmt}_j$ variables for each server to a 64-bit number generated by their RNG upon reboot. For the protocol described in Figure 5·6, 5·7, which randomizes the 32-bit sub-second granularity of the expected origin timestamp, we also require that the second-level granularity of the client's local time t_c isn't replicated too often within NTP queries.

We suppose NTP is authenticated with MAC of length $2n$. Let $on\mathcal{A}$ be any on-path attacker, and let $\texttt{ts}$ be any transcript involving a maximum of s trusted servers per client and a maximum of τ exchanges involving any single client-server pair that replicate any t_c value (up to the second) at most γ times. Let i^* be a client who does not query $on\mathcal{A}$ as server.

As before, we use a sequence of games to prove that $on\mathcal{A}$ tampers the state of $\mathcal{P}_{i^*}$ with probability at most $\epsilon_{on\mathcal{A}}$. We start by reusing games G_0 and G_1 from the proof of Theorem 1 above. It is straightforward to validate that the reduction between those games continues to hold against an on-path attacker. We now build a new sequence of games that (1) reflects $on\mathcal{A}$'s ability to view the contents of honest parties' messages and (2) utilizes the MAC tag to limit $on\mathcal{A}$'s spoofing capacity.

Foreshadowing the end of the proof, we will use Lemma 1 to arrive at the final bound. As such, our analysis simply describes the impact of each game on the probability of success during an NTP exchange (Q_d) and between NTP exchanges (Q_b).

Game G_2'. This game is identical to G_1, except that the network $\mathcal{N}$ is additionally instructed to drop all the packets sent by $on\mathcal{A}$ that are simply 'preplays' of packets

sent between honest parties; *i.e.,* spoofed packets sent by $on\mathcal{A}$ that are identical to existing packets in $\mathcal{N}$'s queue such that $on\mathcal{A}$'s packet will be delivered first.

Recall that our definition of soundness is agnostic to preplay attacks. Hence, forbidding them has no effect on the adversary's success probability, *i.e.,* $\Pr^{1}_{on\mathcal{A}} = \Pr^{2'}_{on\mathcal{A}}$.

Game G'_3. This game is identical to G'_2, except that we abort the execution if the client $\mathcal{P}_{i^*}$ sends two different queries to the same server with identical origin timestamps. We stress that this constraint is independent of $on\mathcal{A}$'s behavior.

Consider a single NTP exchange between client $\mathcal{P}_{i^*}$ and server $\mathcal{P}_j$ where the client's clock begins at t_c. $\mathcal{P}_{i^*}$'s origin timestamp replicates a previous choice with probability at most $q_{32} = 2^{-32}\gamma$ if only the sub-second granularity is randomized or $q_{64} = 2^{-64}\tau$ if the entire expected origin timestamp is randomized. If the honest client repeats an origin timestamp, then $on\mathcal{A}$ may trivially attack an NTP exchange by replaying (an already-MAC'd) responses from previous exchanges.

Hence, the transformation from game G'_2 to game G'_3 affects Q_d by at most q_E. We remark that $on\mathcal{A}$ only requires 1 packet to perform this attack, so the resulting probability is independent of the bandwidth R.

Game G'_4. This game is identical to G'_3, except that the network $\mathcal{N}$ is instructed to drop all of $on\mathcal{A}$'s 'replayed' packets, *i.e.,* packets sent by $on\mathcal{A}$ that are identical to prior packets sent between honest parties and (1) have already been delivered or (2) are in $\mathcal{N}$'s queue for delivery before $on\mathcal{A}$'s packet. These replayed packets will have valid MAC tags but stale origin timestamps.

Consider what happens when a response packet from server $\mathcal{P}_j$ is replayed to target $\mathcal{P}_{i^*}$, or when a query packet from target $\mathcal{P}_{i^*}$ is replayed to server $\mathcal{P}_j$ and elicits $\mathcal{P}_j$'s legitimate response packet. If $\mathcal{P}_{i^*}$ and $\mathcal{P}_j$ are currently engaged in an NTP exchange, then $\mathcal{P}_{i^*}$'s state variable $\texttt{xmt}_j$ is set to an origin timestamp that is distinct

from the one in the replay packet, so the replayed packet definitely fails TEST2 by the constraint imposed by Game G'_3. On the other hand, if $\mathcal{P}_{i^*}$ and $\mathcal{P}_j$ are between exchanges, then $\texttt{xmt}_j$ is set to a randomized value with 64 bits of entropy. Hence, the transformation from game G'_3 to game G'_4 affects Q_b by at most 2^{-64}.

Game G'_5. This game is identical to G'_4, except that the network $\mathcal{N}$ consciously corrupts all MAC tags in *off*$\mathcal{A}$'s spoofed packets that aren't replays or preplays, so they never verify. We note that *on*$\mathcal{A}$ has no chance of winning game G'_5 (that is, $\Pr^{5'}_{on\mathcal{A}} = 0$, and thus $Q_d = Q_b = 0$ for game G'_5) because all of the packets she sends are rejected by their recipients for having invalid tags. Hence, *on*$\mathcal{A}$ cannot get any honest party to read its spoofed packets, much less change their state as a result of them. Additionally, we claim that the Game $G'_4 \to G'_5$ transformation affects Q_d by $\frac{R\delta}{360+n} \cdot \mathcal{A}(\text{EU-CMA})$ and Q_b by $2^{-64} \cdot \mathcal{A}(\text{EU-CMA})$.

To prove the claim, we replace the tags of all *on*$\mathcal{A}$'s packets toward server $\mathcal{P}_j$ or client $\mathcal{P}_{i^*}$ that aren't replays or preplays with an invalid tag $\perp$. We do this one packet at a time, starting with the final packet and working our way back up to the first one. By a simple hybrid argument, we see that each change has an impact with probability at most $\mathcal{A}(\text{EU-CMA})$.

During an exchange, a single forged MAC permits the attacker to respond to a query with timing data of her own choosing, and a simple union bound gives the bound on Q_d stated above. In between exchanges, a forged message must also include the origin timestamp matching $\mathcal{P}_{i^*}$'s randomly-chosen $\texttt{xmt}_j$ or else the packet will fail TEST2, yielding the bound on Q_b.

Putting it all together. Game G_1 additively impacts $\epsilon_{on\mathcal{A}}$, and Game G'_2 has no effect. Games G'_3, G'_4, and G'_5 all depend on k, and they detail the combined vulnerability

of NTP to an on-path attacker during and between exchanges:

$$Q_d \leq q_E + \frac{R\delta}{360 + n} \cdot \mathcal{A}(\text{EU-CMA})$$
$$Q_b \leq 2^{-64} \cdot [1 + \mathcal{A}(\text{EU-CMA})] \approx 2^{-64},$$

where the final approximation follows from the fact that $1 + \mathcal{A}(\text{EU-CMA}) \approx 1$ for any reasonable MAC. Lemma 1 then bounds the probability that $on\mathcal{A}$ affects a particular client-server state k times in a row beginning from a specified starting point, and (as before) multiplying this value by the $s\tau$ possible starting points yields the bound in Theorem 2.

References

"2016 Dyn cyberattack" (2016). 2016 dyn cyberattack. https://en.wikipedia.org/wiki/2016_Dyn_cyberattack (Accessed: July 2019).

Achenbach, D., Müller-Quade, J., and Rill, J. (2015). Synchronous universally composable computer networks. In *Cryptography and Information Security in the Balkans, (BalkanCryptSec'15)*, pages 95–111.

Ager, B., Mühlbauer, W., Smaragdakis, G., and Uhlig, S. (2010). Comparing DNS resolvers in the wild. In *Proceedings of the 10th ACM SIGCOMM Internet Measurement Conference, IMC 2010, Melbourne, Australia - November 1-3, 2010*, pages 15–21.

Akamai (2018). https://www.akamai.com/uk/en/resources/server-load-balancing.jsp (Accessed: July 2019).

Amazon (2015). Simple storage service (s3): Signing and authenticating rest requests. http://docs.aws.amazon.com/AmazonS3/latest/dev/RESTAuthentication.html (Accessed Aug 2015).

Andrews, M. P. (1998). *Negative Caching of DNS Queries (DNS NCACHE)*. Internet Engineering Task Force (IETF). https://doi.org/10.17487/RFC2308.

Archlinux (2018). Network time protocol daemon. https://wiki.archlinux.org/index.php/Network_Time_Protocol_daemon (Accessed: July 2018).

Arends, R., Austein, R., Larson, M., Massey, D., and Rose, S. (2005a). *RFC 4033: DNS Security Introduction and Requirements*. Internet Engineering Task Force (IETF). http://tools.ietf.org/html/rfc4033.

Arends, R., Austein, R., Larson, M., Massey, D., and Rose, S. (2005b). *RFC 4034: Resource Records for the DNS Security Extensions*. Internet Engineering Task Force (IETF). http://tools.ietf.org/html/rfc4034.

Arends, R., Austein, R., Larson, M., Massey, D., and Rose, S. (2005c). *RFC 4035: Resource Records for the DNS Security Extensions*. Internet Engineering Task Force (IETF). http://tools.ietf.org/html/rfc4035.

Atkins, D. and Austein, R. (2004). *Threat Analysis of the Domain Name System (DNS)*. Internet Engineering Task Force (IETF). https://tools.ietf.org/html/rfc3833.

Autokey (2012). Autokey Configuration for NTP stable releases. The NTP Public Services Project: http://support.ntp.org/bin/view/Support/ConfiguringAutokey (Accessed: July 2015).

Axel K (2015). ntpd access restrictions, section 6.5.1.1.3. allow queries? http://support.ntp.org/bin/view/Support/AccessRestrictions.

Backes, M., Manoharan, P., and Mohammadi, E. (2014). Tuc: Time-sensitive and modular analysis of anonymous communication. In *IEEE Computer Security Foundations Symposium*, pages 383–397.

Baggett, M. (2012). IP fragment reassembly with scapy. *SANS Institute InfoSec Reading Room*.

Barker, E. and Roginsky, A. (2011). Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. *NIST Special Publication. http://csrc.nist.gov/publications/nistpubs/800-131A/sp800-131A.pdf*, 800:131A.

Bellare, M., Canetti, R., and Krawczyk, H. (1996). Keying hash functions for message authentication. In *Advances in Cryptology, CRYPTO 1996*, pages 1–15. Springer.

Bhatti, S. N. and Atkinson, R. J. (2011). Reducing dns caching. *2011 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 792–797.

Bicknell, L. (2012). NTP issues today. Outages Mailing List http://mailman.nanog.org/pipermail/nanog/2012-November/053449.html.

Bind (2018). Bind. https://www.isc.org/downloads/bind/ (Accessed: July 2019).

BIPM (2018). What time is it? https://www.bipm.org/en/bipm-services/timescales/time-server.html (Accessed: July 2019).

Borgolte, K., Fiebig, T., Hao, S., Kruegel, C., and GiovanniVigna (2018). Cloud strife: Mitigating the security risks of domain-validated certificates. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*.

Boulakhrif, H. (2015). Analysis of DNS Resolver Performance Measurements. Master's thesis, University of Amsterdam, the Netherlands.

BUGTRAQ mailing list (1997). Linux and Windows IP fragmentation (Teardrop) bug. http://insecure.org/sploits/linux.fragmentation.teardrop.html.

Buldas, A., Laud, P., Saarepera, M., and Willemson, J. (2005). *Universally Composable Time-Stamping Schemes with Audit*, pages 359–373. Springer Berlin Heidelberg, Berlin, Heidelberg.

Canetti, R. (2001). Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE.

Canetti, R. (2004). Universally composable signature, certification, and authentication. In *IEEE Computer Security Foundations Workshop*, page 219. IEEE Computer Society.

Canetti, R. (2013). Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067 (2013 version). https://eprint.iacr.org/2000/067/20130717:020004.

Canetti, R., Dodis, Y., Pass, R., and Walfish, S. (2007). Universally composable security with global setup. In *Theory of Cryptography*, pages 61–85.

Canetti, R., Shahaf, D., and Vald, M. (2016). Universally composable authentication and key-exchange with global PKI. In *Public-Key Cryptography, Proceedings, Part II*, volume 9615 of *Lecture Notes in Computer Science*, pages 265–296. Springer.

Checkpoint (2018). IP fragments (UTM-1 appliance). https://www.checkpoint.com/smb/help/utm1/8.1/2032.htm (Accessed: July 2019).

Chiu, A. (2015). Cisco identifies multiple vulnerabilities in network time protocol daemon (ntpd). TALOS Blog http://blog.talosintel.com/2015/10/ntpd-vulnerabilities.html.

chrony (2015). chrony. https://chrony.tuxfamily.org/ (Accessed: July 2019).

chronyd (2015). https://github.com/mlichvar/chrony/blob/master/ntp_core.c#L908.

Clayton, R., Murdoch, S. J., and Watson, R. N. (2006). Ignoring the great firewall of china. In *Privacy Enhancing Technologies*, pages 20–35. Springer.

"clockgettime" (2018). Posix clock_gettime(). http://pubs.opengroup.org/onlinepubs/9699919799/functions/clock_gettime.html (Accessed: July 2018).

Cloudflare (2018). https://www.cloudflare.com/load-balancing/ (Accessed: July 2019).

Comodo (2011). Fraud incident. https://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html.

Cooper, D., Heilman, E., Brogle, K., Reyzin, L., and Goldberg, S. (2013). On the risk of misbehaving RPKI authorities. *HotNets XII*.

Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and Polk, W. T. (2008). *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. Internet Engineering Task Force (IETF). https://doi.org/10.17487/RFC5280.

Corbixgwelt (2011). Timejacking & bitcoin: The global time agreement puzzle (culubas blog). http://culubas.blogspot.com/2011/05/timejacking-bitcoin_802.html (Accessed Aug 2015).

CVE (2019). Common vulnerabilities and exposures. https://cve.mitre.org/ (Accessed: Aug 2019).

Czyz, J., Kallitsis, M., Gharaibeh, M., Papadopoulos, C., Bailey, M., and Karir, M. (2014a). Taming the 800 pound gorilla: The rise and decline of NTP DDoS attacks. In *Proceedings of the 2014 Internet Measurement Conference*, pages 435–448. ACM.

Czyz, J., Kallitsis, M., Gharaibeh, M., Papadopoulos, C., Bailey, M., and Karir, M. (2014b). Taming the 800 pound gorilla: The rise and decline of NTP DDoS attacks. In *Internet Measurement Conference*, pages 435–448.

Dai, T., Shulman, H., and Waidner, M. (2016). DNSSEC misconfigurations in popular domains. In *Cryptology and Network Security - 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings*, pages 651–660.

DeGroote, T. (2013). Lies, damn lies and dns performance statistics. https://secure64.com/lies-damn-lies-dns-performance-statistics/ (Accessed: July 2019).

d'Itri, M. (2015). Hacking team and a case of bgp hijacking. http://blog.bofh.it/id_456.

Dnsmasq (2018). Dnsmasq. http://www.thekelleys.org.uk/dnsmasq/doc.html (Accessed: July 2019).

dnsmasq (2018). How to dnsmasq. https://wiki.debian.org/HowTo/dnsmasq#Local_Caching (Accessed: July 2019).

Dowling, B., Stebila, D., and Zaverucha, G. (2016). Authenticated network time synchronization. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 823–840, Austin, TX. USENIX Association.

DropBox (2015). Core API. https://www.dropbox.com/developers/core/docs (Accessed Aug 2015).

Duan, H., Weaver, N., Zhao, Z., Hu, M., Liang, J., Jiang, J., Li, K., and Paxson, V. (2012). Hold-on: Protecting against on-path dns poisoning. In *Proc. Workshop on Securing and Trusting Internet Names, SATIN*.

Durairajan, R., Mani, S. K., Sommers, J., and Barford, P. (2015). Time's forgotten: Using ntp to understand internet latency. *HotNets'15*.

Durumeric, Z., Wustrow, E., and Halderman, J. A. (2013). Zmap: Fast internet-wide scanning and its security applications. In *USENIX Security*, pages 605–620. Citeseer.

Eckersley, P. and Burns, J. (2010). An observatory for the SSLiverse. DEFCON'18.

Elz, R. and Bush, R. (1997). *Clarifications to the DNS Specification*. Internet Engineering Task Force (IETF). https://doi.org/10.17487/RFC2181.

Franke, D., Sibold, D., and K. Teichel, M. Dnsarie, R. S. (2018). *draft-ietf-ntp-using-nts-for-ntp-19: Network Time Security for the Network Time Protocol*. Internet Engineering Task Force (IETF). https://datatracker.ietf.org/doc/draft-ietf-ntp-using-nts-for-ntp/.

gentoo linux (2018). System time. https://wiki.gentoo.org/wiki/System_time (Accessed: July 2018).

Gilad, Y. and Herzberg, A. (2013). Fragmentation considered vulnerable. *ACM Trans. Inf. Syst. Secur*, 15(4).

Goldberg, S. (2014). Why is it taking so long to secure internet routing? *Communications of the ACM: ACM Queue*, 57(10):56–63.

Goldreich, O. (2006). *Concurrent Zero-Knowledge with Timing, Revisited*, pages 27–87. Springer Berlin Heidelberg, Berlin, Heidelberg.

Gont, F. (2010). *RFC 5927: ICMP attacks on TCP*. Internet Engineering Task Force (IETF). https://tools.ietf.org/html/rfc5927.

Gordon, J. W. (2016). The day ddos attacks broke the internet. https://www.tektonikamag.com/index.php/2016/10/28/the-day-ddos-attacks-broke-the-internet/ (Accessed: July 2019).

Haber, S. and Stornetta, W. S. (1991). How to time-stamp a digital document. *J. Cryptology*, 3(2):99–111.

Haberman, B. and Mills, D. (2010). *RFC 5906: Network Time Protocol Version 4: Autokey Specification*. Internet Engineering Task Force (IETF). http://tools.ietf.org/html/rfc5906.

Hammer-Lahav, E. (2010). *RFC 5849: The OAuth 1.0 Protocol*. Internet Engineering Task Force (IETF). https://tools.ietf.org/html/rfc5849.

Heilman, E., Cooper, D., Reyzin, L., and Goldberg, S. (2014). From the consent of the routed: Improving the transparency of the RPKI. *ACM SIGCOMM'14*.

Heninger, N., Durumeric, Z., Wustrow, E., and Halderman, J. A. (2012). Mining your ps and qs: Detection of widespread weak keys in network devices. In *USENIX Security Symposium*, pages 205–220.

Herzberg, A. and Shulman, H. (2012a). Antidotes for DNS poisoning by off-path adversaries. In *Seventh International Conference on Availability, Reliability and Security, Prague, ARES 2012, Czech Republic, August 20-24, 2012*, pages 262–267.

Herzberg, A. and Shulman, H. (2012b). Fragmentation considered poisonous. *CoRR*.

Herzberg, A. and Shulman, H. (2013). Fragmentation considered poisonous, or: One-domain-to-rule-them-all. org. In *Communications and Network Security (CNS), 2013 IEEE Conference on*, pages 224–232. IEEE.

Herzberg, A., Shulman, H., and Crispo, B. (2014). Less is more: cipher-suite negotiation for DNSSEC. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 346–355. ACM.

Hodges, J., Jackson, C., and Barth, A. (2012). *HTTP Strict Transport Security (HSTS)*. Internet Engineering Task Force (IETF). https://tools.ietf.org/html/rfc6797.

Holz, T., Gorecki, C., Rieck, K., and Freiling, F. C. (2008). Measuring and detecting fast-flux service networks. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*.

Honeynet (2018). The honeynet project, know your enemy: fast-flux service networks. http://www.honeynet.org/book/export/html/130 (Accessed: July 2018).

Hubert, A. and von Mook, R. (2009). *Measures for Making DNS More Resilient against Forged Answers*. Internet Engineering Task Force (IETF). https://tools.ietf.org/html/rfc5452.

III, D. E. E. and Andrews, M. P. (2016). *Domain Name System (DNS) Cookies*. Internet Engineering Task Force (IETF). https://doi.org/10.17487/RFC7873.

Itkin, E. and Wool, A. (2016). A security analysis and revised security extension for the precision time protocol. *CoRR*, abs/1603.00707.

Jung, J., Sit, E., Balakrishnan, H., and Morris, R. T. (2001). DNS performance and the effectiveness of caching. In *Proceedings of the 1st ACM SIGCOMM Internet Measurement Workshop, IMW 2001, San Francisco, California, USA, November 1-2, 2001*, pages 153–167.

Kalai, Y. T., Lindell, Y., and Prabhakaran, M. (2005). Concurrent general composition of secure protocols in the timing model. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 644–653, New York, NY, USA. ACM.

Kaminsky, D. (2008). It's the end of the cache as we know it. In: Black Hat Conference. `http://www.blackhat.com/presentations/bh-jp-08/bh-jp-08-Kaminsky/BlackHat-Japan-08-Kaminsky-DNS08-BlackOps.pdf` (Accessed: July 2018).

Kareem, K. (2017). TTL override by ISPs – is there a performance cost? `https://dyn.com/blog/ttl-override-by-isps-is-there-a-performance-cost/` (Accessed: July 2018).

Katz, J., Maurer, U., Tackmann, B., and Zikas, V. (2013a). Universally composable synchronous computation. In *TCC*, pages 477–498.

Katz, J., Maurer, U., Tackmann, B., and Zikas, V. (2013b). *Universally Composable Synchronous Computation*, pages 477–498. Springer Berlin Heidelberg, Berlin, Heidelberg.

Kiyawat, K. (2014). Do web browsers obey best practices when validating digital certificates? Master's thesis, Northeastern University.

Klein, A. (2018). Microsoft windows dns stub resolver cache poisoning (ms08-020). `http://www.securiteam.com/securityreviews/5QP022KO1E.html` (Accessed: July 2018).

Klein, A., Shulman, H., and Waidner, M. (2017). Internet-wide study of DNS cache injections. In *2017 IEEE Conference on Computer Communications, INFOCOM 2017, Atlanta, GA, USA, May 1-4, 2017*, pages 1–9.

Klein, J. (2013). Becoming a time lord - implications of attacking time sources. Shmoocon Firetalks 2013: `https://youtu.be/XogpQ-iA6Lw`.

Knockel, J. and Crandall, J. R. (2014). Counting packets sent between arbitrary internet hosts. In *4th USENIX Workshop on Free and Open Communications on the Internet (FOCI'14)*.

knot (2018). Knot resolver. https://www.knot-resolver.cz/ (Accessed: July 2019).

Knowles, B. (2004). NTP support web: Section 5.3.3. upstream time server quantity. http://support.ntp.org/bin/view/Support/SelectingOffsiteNTPServers#Section_5.3.3. (Accessed July 2015).

Kohl, J. and Neuman, C. (1993). *RFC 1510: The Kerberos Network Authentication Service (V5)*. Internet Engineering Task Force (IETF). https://tools.ietf.org/html/rfc1510.

Kolkman, O., Mekking, W., and Gieben, R. (2012). *RFC 6781: DNSSEC Operational Practices, Version 2*. Internet Engineering Task Force (IETF). http://tools.ietf.org/html/rfc6781.

Kolkman, O. M. and Gieben, R. M. (2006). *DNSSEC Operational Practices*. Internet Engineering Task Force (IETF). https://doi.org/10.17487/RFC4641.

Krämer, L., Krupp, J., Makita, D., Nishizoe, T., Koide, T., Yoshioka, K., and Rossow, C. (2015). Amppot: Monitoring and defending against amplification ddos attacks. In *International Workshop on Recent Advances in Intrusion Detection*, pages 615–636. Springer.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565.

Langley, A. (2014). Revocation still doesn't work. https://www.imperialviolet.org/2011/03/18/revocation.html.

Larsen, M. and Gont, F. (2011). *RFC 6056: Recommendations for Transport-Protocol Port Randomization*. Internet Engineering Task Force (IETF). https://tools.ietf.org/html/rfc6056.

Lawrence, D. and Kumari, W. (2017). *Serving Stale Data to Improve DNS Resiliency*. Internet Engineering Task Force (IETF). https://tools.ietf.org/html/draft-ietf-dnsop-serve-stale-00.

Leiner, B. M., Cerf, V. G., Clark, D. D., Kahn, R. E., Kleinrock, L., Lynch, D. C., Postel, J. B., Roberts, L. G., and Wolff, S. S. (1999). A brief history of the internet. *CoRR*, cs.NI/9901011.

Lepinski, M. and Kent, S. (2012). *RFC 6480: An Infrastructure to Support Secure Internet Routing*. Internet Engineering Task Force (IETF). https://tools.ietf.org/html/rfc6480.

Linux (2018). Linux system administrators guide. https://www.tldp.org/LDP/sag/html/hw-sw-clocks.html (Accessed: July 2018).

Love, R. (2013). *Linux System Programming*. O'Reilly Media, Inc.

Malhotra, A., Cohen, I. E., Brakke, E., and Goldberg, S. (2016). Attacking the network time protocol. In *Network and Distributed System Security Symposium*.

Malhotra, A. and Goldberg, S. (2016). Attacking NTP's Authenticated Broadcast Mode. *SIGCOMM Computer Communication Review*, pages 12–17.

Malhotra, A. and Goldberg, S. (2019). *RFC 8573: Message Authentication Code for the Network Time Protocol*. Internet Engineering Task Force (IETF). `https://tools.ietf.org/html/rfc8573`.

Malhotra, A., Gundy, M. V., Varia, M., Kennedy, H., Gardner, J., and Goldberg, S. (2017). The security of NTP's datagram protocol. In *Financial Cryptography and Data Security - 21st International Conference, FC, Malta*, pages 405–423.

Mamakos, L., Lidl, K., Evarts, J., Carrel, D., Simone, D., and Wheeler, R. (1999). *RFC 2516: A Method for Transmitting PPP Over Ethernet (PPPoE)*. Internet Engineering Task Force (IETF). `https://tools.ietf.org/html/rfc2516`.

Marlinspike, M. (2009). New tricks for defeating ssl in practice. *BlackHat DC, February*.

Marzullo, K. A. (1984). *Maintaining the Time in a Distributed System*. PhD thesis, Stanford.

Matsuo, T. and Matsuo, S. (2005). On universal composable security of time-stamping protocols. In *Conference on Applied Public Key Infrastructure: 4th International Workshop, IWAP*, pages 169–181, Amsterdam, The Netherlands, The Netherlands. IOS Press.

Mauch, J. (2015). openntpproject: NTP Scanning Project. `http://openntpproject.org/`.

Mauch, J. (2018). Open resolver project. `http://openresolverproject.org/` (Accessed: July 2018).

Menscher, D. (2012). NTP issues today. Outages Mailing List `http://mailman.nanog.org/pipermail/nanog/2012-November/053494.html`.

Micali, S. (2016). ALGORAND: the efficient and democratic ledger. *CoRR*, abs/1607.01341.

Microsoft (2010). MS05-019: Vulnerabilities in TCP/IP could allow remote code execution and denial of service. `https://support.microsoft.com/en-us/kb/893066` (Accessed July 2015).

Miller, I. (2001). *RFC 3128 (Informational): Protection Against a Variant of the Tiny Fragment Attack*. Internet Engineering Task Force (IETF). https://tools.ietf.org/html/rfc3128.

Mills, D. (1992). *RFC 1305: Network Time Protocol (Version 3) Specification, Implementation and Analysis*. Internet Engineering Task Force (IETF). http://tools.ietf.org/html/rfc1305.

Mills, D. (2006). *RFC 4330: Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI*. Internet Engineering Task Force (IETF). https://tools.ietf.org/html/rfc4330.

Mills, D. (2014). Association management. https://www.eecis.udel.edu/~mills/ntp/html/assoc.html.

Mills, D. and Haberman, B. (2016). *draft-haberman-ntpwg-mode-6-cmds-00: Control Messages Protocol for Use with Network Time Protocol Version 4*. Internet Engineering Task Force (IETF). https://datatracker.ietf.org/doc/draft-haberman-ntpwg-mode-6-cmds/.

Mills, D., Martin, J., Burbank, J., and Kasch, W. (2010). *RFC 5905: Network Time Protocol Version 4: Protocol and Algorithms Specification*. Internet Engineering Task Force (IETF). http://tools.ietf.org/html/rfc5905.

Mills, D. L. (1985). *RFC 958: Network Time Protocol (NTP)*. Internet Engineering Task Force (IETF). https://tools.ietf.org/html/rfc958.

Mills, D. L. (2011). *Computer Network Time Synchronization*. CRC Press, 2nd edition.

Minar, N. (1999). A survey of the NTP network.

Mizrahi, T. (2012a). A game theoretic analysis of delay attacks against time synchronization protocols. In *Precision Clock Synchronization for Measurement Control and Communication (ISPCS)*, pages 1–6. IEEE.

Mizrahi, T. (2012b). *RFC 7384 (Informational): Security Requirements of Time Protocols in Packet Switched Networks*. Internet Engineering Task Force (IETF). http://tools.ietf.org/html/rfc7384.

Mkacher, F., Bestel, X., and Duda, A. (2018). Secure time synchronization protocol. In *IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication, ISPCS 2018, Geneva, Switzerland, September 30 - Oct. 5, 2018*, pages 1–6.

Mockapetris, P. (1987a). *RFC 1034: Domain Names: Concepts and Facilities*. Internet Engineering Task Force (IETF). https://tools.ietf.org/html/rfc1034.

Mockapetris, P. V. (1987b). *Domain names - implementation and specification*. Internet Engineering Task Force (IETF). https://doi.org/10.17487/RFC1035.

Moreira, N., Lazaro, J., Jimenez, J., Idirin, M., and Astarloa, A. (2015). Security mechanisms to protect ieee 1588 synchronization: State of the art and trends. In *Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS), 2015 IEEE International Symposium on*, pages 115–120. IEEE.

Morowczynski, M. (2012). Did your active directory domain time just jump to the year 2000? Microsoft Server & Tools Blogs http://blogs.technet.com/b/askpfeplat/archive/2012/11/19/did-your-active-directory-domain-time-just-jump-to-the-year-2000.aspx.

Murta, C. D., Torres Jr, P. R., and Mohapatra, P. (2006). Characterizing quality of time and topology in a time synchronization network. In *GLOBECOM*.

Mutton, P. (2014). Certificate revocation: Why browsers remain affected by heartbleed. http://news.netcraft.com/archives/2014/04/24/certificate-revocation-why-browsers-remain-affected-by-heartbleed.html.

Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system.

National Vulnerability Database (2014). CVE-2014-9295: Multiple stack-based buffer overflows in ntpd in ntp before 4.2.8. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-9295.

NIST (2010). The NIST authenticated ntp service. http://www.nist.gov/pml/div688/grp40/auth-ntp.cfm (Accessed: July 2015).

NIST (2018). Nist authenticated ntp service. https://www.nist.gov/pml/time-and-frequency-division/time-services/nist-authenticated-ntp-service (Accessed: July 2019).

Novak, J. (2005). Target-based fragmentation reassembly. Technical report, Sourcefire, Incorporated.

ntimed (2015). http://nwtime.org/projects/ntimed/ (Accessed: July 2019).

NTPrand (2014). Ntp-project/ntp. https://github.com/ntp-project/ntp/blob/1a399a03e674da08cfce2cdb847bfb65d65df237/libntp/ntp_random.c (Accessed: July 2019).

NTPsec (2017). Welcome to ntpsec. https://www.ntpsec.org/.

openNTPD (2012). https://github.com/philpennock/openntpd/blob/master/client.c#L174.

OWASP (2015). HTTP Strict Transport Security. The Open Web Application Security Project (OWASP): https://www.owasp.org/index.php/HTTP_Strict_Transport_Security.

Pan, J., Hou, Y. T., and Li, B. (2003). An overview of dns-based server selections in content distribution networks. *Computer Networks*, pages 695–711.

Pappas, V., Ed., and E. Osterweil, E. (2012). *Improving DNS Service Availability by Using Long TTL Values*. Internet Engineering Task Force (IETF). https://tools.ietf.org/id/draft-pappas-dnsop-long-ttl-04.html.

Pass, R. and Shi, E. (2016). The sleepy model of consensus. Cryptology ePrint Archive, Report 2016/918. http://eprint.iacr.org/2016/918.

Perrig, A., Canetti, R., Tygar, J. D., and Song, D. (2005). The tesla broadcast authentication protocol. *RSA CryptoBytes*, 5.

Peterson, A. (2013). Researchers say u.s. internet traffic was re-routed through belarus. that's a problem. *Washington Post Blogs: The Switch*.

Postel, J. (1980). *RFC 768: User Datagram Protocol*. Internet Engineering Task Force (IETF). https://tools.ietf.org/html/rfc768.

Postel, J. (1981a). *RFC 791: INTERNET PROTOCOL: DARPA INTERNET PROGRAM PROTOCOL SPECIFICATION*. Internet Engineering Task Force (IETF). https://tools.ietf.org/html/rfc791.

Postel, J. (1981b). *RFC 792: INTERNET CONTROL MESSAGE PROTOCOL*. Internet Engineering Task Force (IETF). https://tools.ietf.org/html/rfc792.

Powerdns (2018). Powerdns. https://www.powerdns.com/(Accessed:July2019).

Ptacek, T. H. and Newsham, T. N. (1998). Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc.

ptpd (2015). https://sourceforge.net/projects/ptpd2/ (Accessed: July 2019).

Rekhter, Y., Moskowitz, B. G., Karrenberg, D., de Groot, G. J., and Lear, E. (1996). *Address Allocation for Private Internets*. Internet Engineering Task Force (IETF). https://doi.org/10.17487/RFC1918.

Rescorla, E. (2018). *The Transport Layer Security (TLS) Protocol Version 1.3*. Internet Engineering Task Force (IETF). https://tools.ietf.org/html/rfc8446.

ripe (2018). Ripe ncc. https://atlas.ripe.net/ (Accessed: July 2018).

ripe NCC (2018). Dns measurement. https://atlas.ripe.net/measurements/8310237/ (Accessed: July 2019).

rootops (2015). Events of 2015-11-30. http://www.root-servers.org/news/events-of-20151130.txt (Accessed: July 2019).

Röttger, S. (2012). Analysis of the ntp autokey procedures. Master's thesis, Technische Universität Braunschweig.

Röttger, S. (2015). Finding and exploiting ntpd vulnerabilities. http://googleprojectzero.blogspot.co.uk/2015/01/finding-and-exploiting-ntpd.html.

roughtime (2015). https://roughtime.googlesource.com/roughtime (Accessed: July 2019).

Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and Adams, C. (2013). *RFC 6960: X.509 Internet Public Key Infrastructure Online Certificate Status Protocol OCSP*. Internet Engineering Task Force (IETF). https://tools.ietf.org/html/rfc6960.

Schiff, N. R., Schapira, M., Dolev, D., and Deutsch, O. (2018). Preventing (network) time travel with chronos. In *Proceedings of the Applied Networking Research Workshop, ANRW 2018, Montreal, QC, Canada, July 16-16, 2018*, page 17.

Schomp, K., Callahan, T., Rabinovich, M., and Allman, M. (2014). Assessing DNS vulnerability to record injection. In *Passive and Active Measurement - 15th International Conference, PAM 2014, Los Angeles, CA, USA, March 10-11, 2014, Proceedings*, pages 214–223.

Schramm, C. (2012). Why does Linux enforce a minimum MTU of 552? http://cschramm.blogspot.com/2012/12/why-does-linux-enforce-minimum-mtu-of.html.

Selvi, J. (2014). Bypassing HTTP strict transport security. *Black Hat Europe*.

Selvi, J. (2015). Breaking SSL using time synchronisation attacks. *DEFCON'23*.

Shankar, U. and Paxson, V. (2003). Active mapping: Resisting NIDS evasion without altering traffic. In *Symposium on Security and Privacy*, pages 44–61. IEEE.

Sherman, J. A. and Levine, J. (2016). Usage analysis of the NIST internet time service. *Journal of Research of the National Institute of Standards and Technology*, 121:33.

Sibold, D., Roettger, S., and Teichel, K. (2015). *draft-ietf-ntp-network-time-security-10: Network Time Security*. Internet Engineering Task Force (IETF). https://tools.ietf.org/html/draft-ietf-ntp-network-time-security-10.

Singh, K. and Schulzrinne, H. (2007). Failover, load sharing and server architecture in SIP telephony. *Computer Communications*, pages 927–942.

SiteUptime (2016). The dyn ddos attack that broke the internet: Here's what happened. https://www.siteuptime.com/blog/2016/12/05/the-dyn-ddos-attack-2016/ (Accessed: July 2019).

Snijders, J. (2018). Nlnog ring. https://ring.nlnog.net/ (Accessed: July 2019).

Steinberg, J. (2016). What we know about friday's cyber attack that broke the internet. https://www.inc.com/joseph-steinberg/what-we-know-about-fridays-cyber-attack-that-broke-the-internet.html (Accessed: July 2019).

Stenn, H. (2015a). Antw: Re: Proposed REFID changes. NTP Working Group Mailing List http://lists.ntp.org/pipermail/ntpwg/2015-July/002291.html.

Stenn, H. (2015b). Ntf releases ntp security patches in ntp-4.2.8p4. https://www.nwtime.org/ntf-releases-ntp-security-patches-ntp-4-2-8p4/.

Stenn, H. (2015c). NTP's REFID. http://nwtime.org/ntps-refid/.

Stenn, H. (2015d). Securing the network time protocol. *Communications of the ACM: ACM Queue*, 13(1).

Stenn, H. (2016). Security notice. http://support.ntp.org/bin/view/Main/SecurityNotice.

systemd (2018). systemd-resolved. https://www.freedesktop.org/software/systemd/man/systemd-resolved.service.html (Accessed: July 2019).

Tatarinov, M. http://support.ntp.org/bin/view/Main/NtpBug2952.

time (2018). Time. https://wiki.archlinux.org/index.php/time (Accessed: July 2018).

Turner, S. and Chen, L. (2011). *RFC 6151:Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms*. Internet Engineering Task Force (IETF). https://tools.ietf.org/html/rfc6151.

Unbound (2018). Unbound. https://nlnetlabs.nl/projects/unbound/about/ (Accessed: July 2019).

USNO (2015). DoD Customers : Authenticated NTP. http://www.usno.navy.mil/USNO/time/ntp/dod-customers (Accessed: July 2015).

Vajda, I. (2016). On the analysis of time-aware protocols in universal composability framework. *International Journal of Information Security*, 15(4):403–412.

Vixie, P. (2014). Rate-limiting state. *Communications of the ACM: ACM Queue*, 12(2):10.

Vixie, P. and Schryver, V. (2012). DNS Response Rate Limiting (DNS RRL). `http://ss.vix.su/~vixie/isc-tn-2012-1.txt`.

VMware (2011). Timekeeping in vmware virtual machines: vsphere 5.0, workstation 8.0, fusion 4.0. `http://www.vmware.com/files/pdf/Timekeeping-In-VirtualMachines.pdf` (Accessed July 2015).

Wang, X. and Yu, H. (2005). How to break MD5 and other hash functions. In *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, pages 19–35.

Weaver, N., Sommer, R., and Paxson, V. (2009). Detecting forged TCP reset packets. In *NDSS*.

wiki, S. (2018). System time. `https://en.wikipedia.org/wiki/System_time` (Accessed: July 2018).

Yu, Y., Wessels, D., Larson, M., and Zhang, L. (2012). Authority server selection in DNS caching resolvers. *Computer Communication Review*.

Yuan, L., Kant, K., Mohapatra, P., and Chuah, C. (2006). Dox: A peer-to-peer antidote for dns cache poisoning attacks. In *2006 IEEE International Conference on Communications*, pages 2345–2350.

Zhang, L., Choffnes, D., Levin, D., Dumitras, T., Mislove, A., Schulman, A., and Wilson, C. (2014). Analysis of SSL certificate reissues and revocations in the wake of heartbleed. In *Internet Measurement Conference (IMC'14)*, pages 489–502.

CURRICULUM VITAE